AD-A067 746 ARIZONA UNIV TUCSON DEPT OF CHEMISTRY F/G 9/2

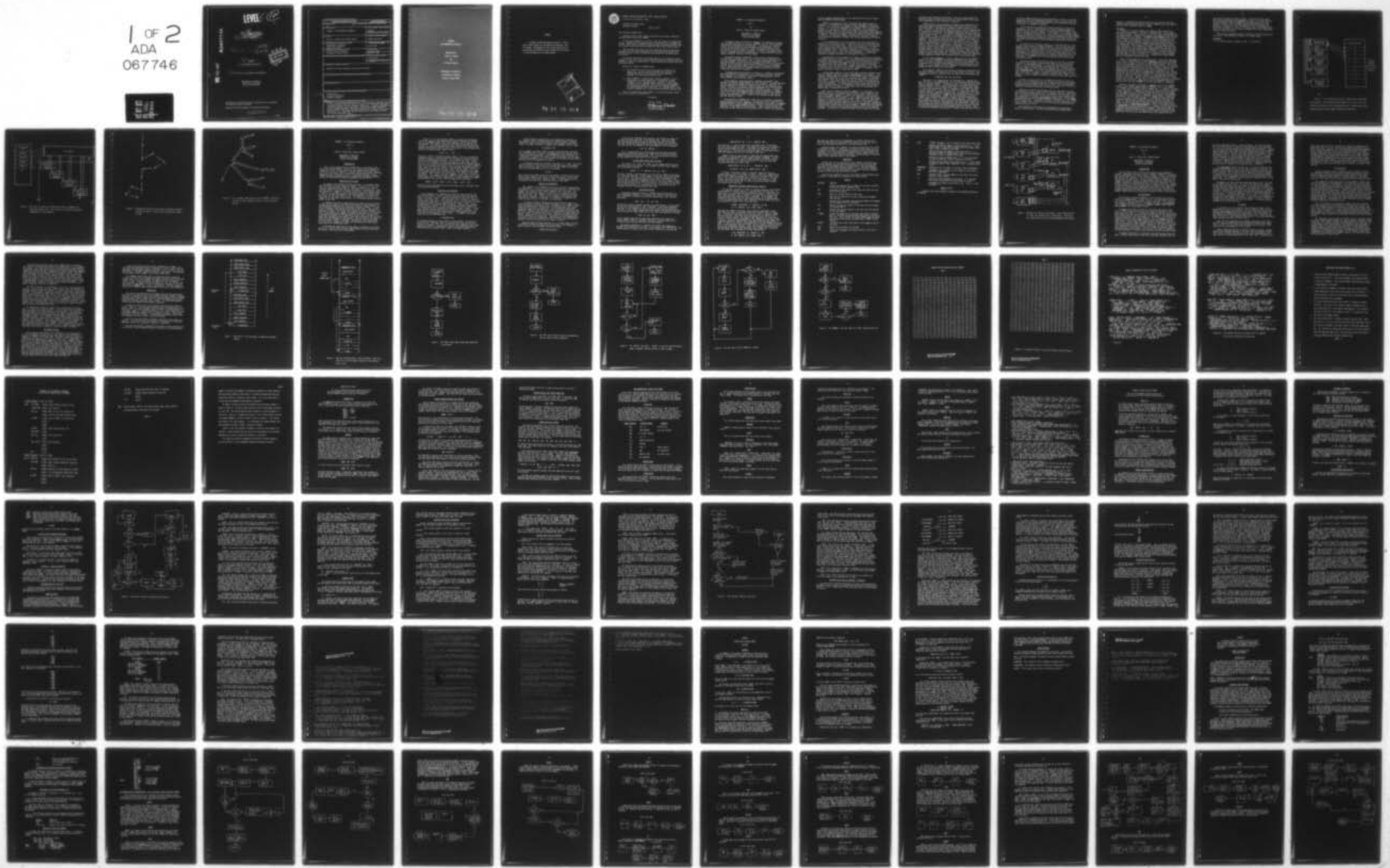
CONVERS: AN INTERPRETIVE COMPILER.(U)

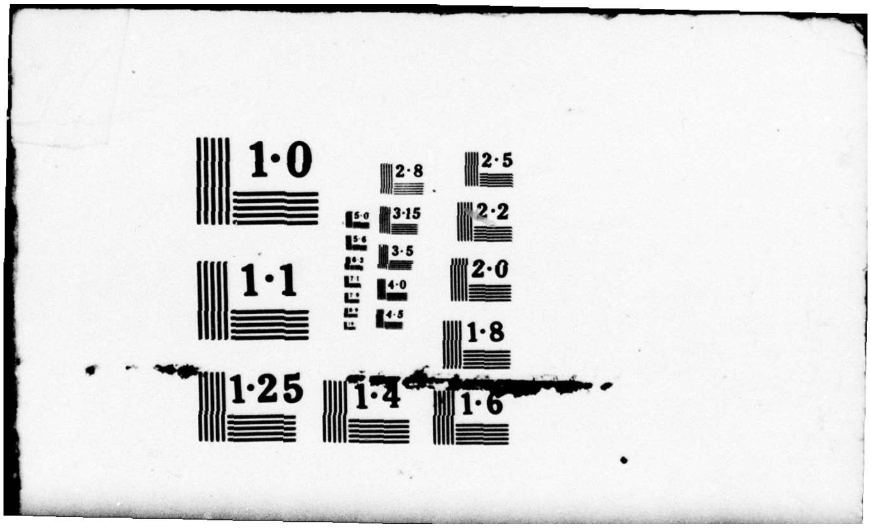
APR 79 S B TILDEN, M B DENTON N00014-75-C-0513

UNCLASSIFIED TR-19 NL

1 OF 2

ADA 067746




1·0
2·8
2·5
5·0
3·15
2·2
3·5
1·1
2·0
4·0
4·5
1·8
1·25
1·4
1·6

15

OFFICE OF NAVAL RESEARCH
Contract N00014-75-C-0513
Task No. NR 051-549
9 Technical Report No. 19

11 4 Apr 79

12 244p.

14 TR-19

6

CONVERS:
AN INTERPRETIVE COMPILER.

10 Scott B. Tilden and M. Bonner Denton

Department of Chemistry
University of Arizona
Tucson, Arizona 85721

033860

LB

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 19	2. GOVT ACCESSION NO.
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) CONVERS: AN INTERPRETIVE COMPILER	5. TYPE OF REPORT & PERIOD COVERED INTERIM
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Scott B. Tilden and M. Bonner Denton	8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0513
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Chemistry University of Arizona Tucson, AZ 85721	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 051-549
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, Virginia 22217	12. REPORT DATE April 4, 1979
	13. NUMBER OF PAGES 61
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

CONVERS Manual
Interpretive Compiler
Computer Language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

CONVERS, a new software package based on an "interpretive compiler" concept, is described which provides a means of rapidly and efficiently developing customized software for any individual application. Numerous advantages are realized compared to other approaches, including high speed operation, superior memory efficiency, stack manipulation and a variety of high level constructs, etc. The unique concepts employed are contrasted with more conventional approaches.

CONVERS

AN INTERPRETIVE COMPILER

Developed by:

Scott B. Tilden

and

M. Bonner Denton

Department of Chemistry
University of Arizona
Tucson, Arizona 85721

CONVERS

A software system developed by Scott B. Tilden and M. Bonner Denton at the Chemistry Department, Univ. of Arizona. Development of this system was partially supported by the Office of Naval Research and an Alfred P. Sloan Fellowship to M. Bonner Denton.

THE UNIVERSITY OF ARIZONA
TUCSON, ARIZONA 85721

COLLEGE OF LIBERAL ARTS
DEPARTMENT OF CHEMISTRY

May 10, 1978

Dear Potential CONVERS User,

Enclosed you will find a manual describing the concepts, operation, and 8080 source listing for CONVERS.

As with any "language", the fastest route to fluency is through hands on use. Additionally, in repeated cases, individuals who have brought up CONVERS, report back escalating enthusiasm about the powers inherent within its concepts which they could not appreciate until they started using it.

Since without a question, Scott and I have not found all the errors, your help will be greatly appreciated in both bringing our attention to outright mistakes as well as detailing any portions of the manual which are unclear.

If you have facilities to read paper tape, we will furnish you with a source of both the initial machine code dictionary and standard high level CONVERS (it saves you a great deal of typing and possible problems with the associated errors).

We have two requests of CONVERS users:

(1) Let us know who you are, the type of CPU, peripherals and application - we are starting a CONVERS users group to encourage communication and hopefully develop sufficient standards to allow software sharing, etc.

(2) Since CONVERS is a highly plastic software package, you will without question end up modifying it to generate a custom "language" to fit your application. However, to facilitate recognition of the CONVERS concept behind your custom package, we would greatly appreciate your retaining the word CONVERS and adding prefixes and/or suffixes, i.e., CONVERS 6800 (a 6800 version), CONVERS A.S.U. (Arizona State University version), etc.

We wish you the best of luck; however, if you have problems, please call Scott or myself at 602-884-3758.

Sincerely,

Scott B. Tilden

M Bonner Denton
M. Bonner Denton
Associate Professor

MBD/mc
Enclosure

CONVERS - an Interpretive Compiler

Part I

by

Scott B. Tilden & M. Bonner Denton

Department of Chemistry
University of Arizona
Tucson, Arizona 85721

The common classes of high level languages are interpreters (exemplified by BASIC) and compilers (such as FORTRAN). Each of these approaches have definite advantages and disadvantages. Interpreters are conversational, giving caution and error messages during programming, but are relatively slow during execution. Compilers generally run much faster, but are not as easily edited or debugged. In control environments, both types of languages share the disadvantage of it being difficult to incorporate custom I/O "drivers" into the system.

One obvious question immediately arises--why not incorporate the most desirable characteristics of each of these software approaches into a single language? Additionally, due to unique requirements found in many applications, why not allow the programmer the flexibility to actually develop his own individual modifications and additions to the language itself? Other desirable features would include high memory efficiency, ease of understanding how the language operates and the ability to be transferred from one CPU to another.

A language described originally in 1974 by C. H. Moore at the National Radio Observatory (1) offers many of the advantages described. However, this language, called FORTH^{TM}, has the major disadvantage that its internal workings are exceedingly complex.

Using experience gained with BASIC, FORTRAN, and FORTH^{TM}, a new approach to software, named CONVERS, has been developed. The word "approach" has been used because CONVERS provides the user with the capability of rapidly developing a customized software package for any individual application. A wealth of generalized software programming aids, I/O routines, and high level constructs are provided to make this possible. Through utilizing advanced programming concepts including threaded code, software stacks, etc., the entire CONVERS support system resides in less than 4K bytes on an 8080 based system. An entire disk operating system requires only 800 additional bytes!

The discussion of CONVERS will center first on a general introduction to the concepts behind the language. A second article will deal with programming examples, giving a great deal of attention to the types of 'commands' supported by the standard CONVERS package. Finally, the last article will treat the 'initial machine code dictionary'. These routines are the low lying code on which the rest of the system relies. In this last discussion, several important methodologies will be treated which are used to implement the CONVERS software system.

CONVERS has properties of both interpreter and compiler types of languages. An interpreter has the one obvious advantage of being interactive. This is the case with CONVERS, as soon as CONVERS is loaded and running, it is waiting for commands much like BASIC. This aspect is important; the user should be spared the drudgery of constant assembling, loading of object code, run time packages, even front panel switch toggeling followed by reloading the assembler and source code when additional editing is required. As important, CONVERS interacts with the user by echoing error and warning diagnostic messages both during programming and program execution.

A true interpreter also has the property of actually interpreting symbolic source code that exists in some file area or is being input from a terminal or mass-storage device. With most interpreters, the execution of the symbolic code always remains within the interpreter, i.e. each symbol (command) that is recognized corresponds to a string of code within the interpreter itself. Thus, user program execution is really only directing the execution of code within the interpreter. A conventional interpretive program structure is illustrated in Figure 1.

Disadvantages of this interpretive structure are obviously low execution speeds and lack of flexibility. Each command must first be interpreted followed by command execution. In many cases, the interpretation cycle to identify the command takes much more time than the actual execution of the command within the interpreter. Another contribution to the slow execution of this structure is that commands interpreted in the 'past' must be reinterpreted each time this particular command is reiterated. This process can easily 'waste' a great deal of time.

A conventional compiler, on the other hand, transforms symbolic source code into machine code which executes independently of the compiler. For instance, FORTRAN compilers designed to run on many types of minicomputers (and some microcomputers) will first transform user symbolic source code into assembly code. An assembler must then transform this into machine code which is subsequently loaded and executed, often in conjunction with a "run time package". The program contains within itself all the code it needs to execute properly. The advantage of the compiler is in terms of speed since the user's symbolic source code itself is transformed into machine code, illustrated in Figure 2.

When initially loaded and running, CONVERS is an interpreter, i.e. CONVERS contains all the programming algorithms to input symbolic source code and to properly execute it. When a new algorithm is initially entered (programmed), CONVERS converts it to a completely compiled machine code, or to the starting addresses of other compiled machine code programs necessary to properly execute its function. Thus, at the time when new programs are being defined, CONVERS is acting as an 'interpretive compiler', supplying machine code within the body of the new program while maintaining conversational interactions with the programmer. CONVERS keeps the operator informed of the status of the program by outputting error and diagnostic messages. To execute this compiled program, the user needs only to enter the name of the program on the terminal. Once this program (entry) is located, control is passed totally into this program which begins execution independently of the interpreter subroutine.

CONVERS maintains the advantages of compiler and interpretive language structures by separating the compile and execute modes in time. The execute mode is the mode the interpreter assumes to execute a program. The user (or some mass-storage device, etc.) inputs a program name, the interpreter searches for the program in memory and, once located, it is executed in its entirety. The CONVERS interpreter subroutine can also assume a compiler mode. In this mode, the interpreter is used to compile a new program. To put the interpreter in the compile mode, the user types a colon (:) followed by a space (the space is a universal delimiter in CONVERS). The interpreter assumes that a new program is to be compiled; since all programs must have a name, the user must now input a suitable name for the new program. (Again, as is always the case, a space must follow.) What are legal program names? The answer is simple--any ASCII character string of any length (up to 128 characters) is a valid name for the new program. What follows the program name are the names of previous programs which will define the new program's function. Since the interpreter is in the compile mode, these programs are compiled into the new program instead of being executed. The semicolon (;) ends the compilation of the program and returns the interpreter back to the execute mode.

The new program remains as a totally compiled part of the CONVERS system, i.e. this program can likewise be employed in later programs. There is no need to recompile any program once originally defined. This eliminates one of the major disadvantages of purely interpretative languages, i.e. each command, series of commands, or whole programs must be recompiled each time they are encountered.

As an example, suppose the user wishes to define a program which has the simple function of ringing the bell on the terminal five times. The following program may be defined, with the name of the new program being 'SOUND':

```
: SOUND BELL BELL BELL BELL BELL ;
```

The colon (:) is recognized by the interpreter as a command which puts the system in the compile mode. This is followed by the new program name, in this case 'SOUND'. The program, 'BELL', has been defined in the standard CONVERS system as a trivial program to ring the terminal bell. Since the system is in the compile mode, 'BELL' is compiled into 'SOUND', i.e. the starting address of 'BELL' is deposited into 'SOUND' following a machine code 'call'. Since four more 'BELL's' follow, the above compilation scheme is repeated four more times. The semicolon (;) is recognized by the interpreter as a command to end compilation and return the system to the execute mode. 'SOUND' may now be executed by entering 'SOUND' on the terminal (followed by a space); once the interpreter locates 'SOUND', a 'call' is made to its starting address. Since 'SOUND' has already been compiled, the interpreter can place program control into 'SOUND' which will execute independently of the interpreter. Thus, the only overhead suffered by the program when executed is the time required for one dictionary search and the subsequent calls and returns. Since an efficient programmer would program in assembly using the same calling structure, CONVERS programs execute approximately as fast as is possible using almost any other programming language. Thus, it can be seen that CONVERS gives to the user the advantages of both interpretative and compiler types of programming languages.

This discussion has treated user defined programs as an entity in themselves. In fact, these programs are stored in computer memory as discrete entities in a set fashion. The way programs are stored is analogous to the way words are stored in a dictionary, i.e. the name of the entry is followed by the definition. Due to the similarity between CONVERS programs and dictionary entries, the following terminology has been adopted. The programs making up the CONVERS package will be called dictionary entries, and the dictionary will name the area of memory where entries are stored.

The concept of a dictionary format is, as discussed above, a close analogy to the internal structure of CONVERS. Just like new words are defined using other words, CONVERS defines new entries using previously defined entries. Initially, however, a dictionary must define words uniquely (or the language would go around in circles); the same is true with CONVERS, the low-lying entries are defined using machine code. New words that must be uniquely defined are constantly being added to the dictionary. The CONVERS user can also define new entries at any time using machine code.

The user has access to dictionary entries at any level, including the initial machine code dictionary which makes up the 'interpretive compiler' of the CONVERS system. The fact that the CONVERS user has access to entries that represent any number of levels of definitions from very simple to complex, means that CONVERS uses a type of structure called *threaded code*. In a typical programming structure, the flow of logic might look like Figure 3, assuming no conditional jumps are to be made. The letters A through I represent separate logical steps in the program. A typical CONVERS structure, on the other hand, might look like Figure 4.

In fact, one way to think of the CONVERS dictionary is merely an "errector set" of components which may be used over and over to assemble an ever increasingly sophisticated set of modules which can themselves be employed to build higher level functions etc., any level of which can readily be intermixed to achieve the desired program.

Note that in Figure 4 the program starts at A and ends at A after going through a series of loops to other programs (entries). What advantage does threaded programming have? The answer to this question is simple: each entry can be used in a variety of other entries because each entry represents a logically *self-contained* program unit. In the *linear* programming approach illustrated in Figure 3, the operation of B is dependent on A and so on. It would be difficult to extract, say, Step C, and use it between E and F unless this possibility was envisioned when the *entire* program was written.

The statement has been made that each CONVERS entry represents a separate logical entity. How then do parameters pass between these entries? A threaded code structure that does not allow for efficient parameter passing would be totally useless. CONVERS solves this requirement by using a structure known as a stack.

THE STACK

The stack is an area of memory set aside to handle numbers and other types of parameters. The stack in CONVERS starts at high memory and grows downward. The dictionary, on the other hand, grows upward from the support system. Parameters are pushed on and popped off the stack by manipulating the stack pointer. The stack pointer always points to the last parameter to be pushed on the stack. To pop a parameter off the stack, a copy of the parameter is made into the desired register(s) as the entry is executing while decrementing the stack pointer (actually it is incrementing since the stack "grows" from the top of memory toward lower locations). This effectively 'removes' the number from the stack. Note that while entries normally only remove numbers from the top of the stack, there exist entries that copy a parameter at any level under the stack and push the copy back on top of the stack. It should be obvious that the stack is like a rubberband that is constantly being stretched and contracted as entries are executing. CONVERS passes approximately 98% of all parameters through the stack.

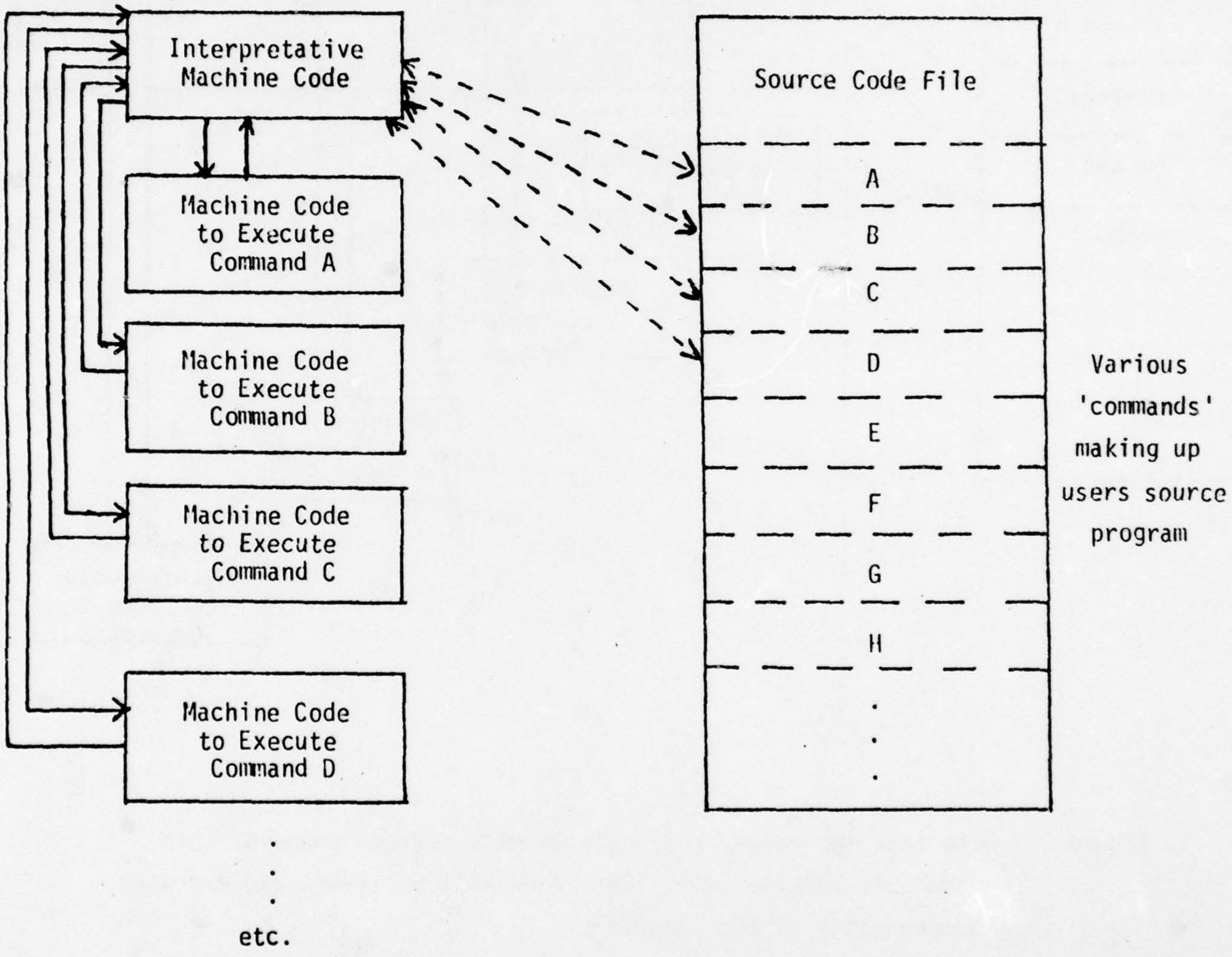
What is the advantage of stack architecture? The primary advantage of the stack is that entries can leave temporary parameters on the stack (to be 'later' popped off and used in other entries) without specifically having to assign memory locations to store these parameters. This advantage is more important than just a savings in memory if specific areas had to be reserved. A given entry does not have to contain addresses for temporary data storage making the entry easily relocatable, i.e. user defined entries can be stored as source code on some sort of mass storage device and later be brought in and compiled; this compiled entry can now be used in later entries without regard as to where parameter storage is located in the entry. However, allowing some parameters to have a fixed location can be useful (in the case of initialization constants or variables). In this case, we create an entry that stores the parameter along with a code that will push the parameter on to the stack when it is executed.

Whenever a new entry is defined, it remains as part of the CONVERS system. Thus, CONVERS itself expands as the user sits at the terminal, i.e. CONVERS does not have a limited 'command' set. A 'command', of course, is any entry that is recognized as part of the dictionary at that moment in time. Any entry is immediately executed by simply typing the entry name followed by a space. This means that one can write, using CONVERS, a system that simulates BASIC, or FORTRAN, or any computer language for that matter. But why should one want to? CONVERS can best be thought of as a 'language' to write languages. In the typical situtation, the 'language' that one would create will execute some task or series of tasks. All the entries that are defined to execute this specific task would be called an application dictionary. One can store these entries as source code on a mass-storage device. If later in time the user wishes to execute this application dictionary, the dictionary would simply be loaded and recompiled automatically by CONVERS. Once the task has been completed, the application dictionary may be removed by the user or the application dictionary can even remove itself while requesting a new application dictionary from mass storage. The ease with which these application dictionaries can be swapped in and out is a direct function of the relocatable nature of CONVERS entries made possible by stack parameter passing. Since the stack and dictionary are separate, one application dictionary can leave parameters for the next application dictionary to operate on. This is, of course, what virtual memory architecture is all about; a small system can behave like a very large system with only speed sacrificed.

This work was supported in part by the Office of Naval Research and an Alfred P. Sloan Research Fellowship to M. Bonner Denton.

REFERENCES

1. C. H. Moore, Astron. Astrophys. Suppl., 15 (1974) 497.



Interpretative Machine Code
Machine Code to Execute Command A
Machine Code to Execute Command B
Machine Code to Execute Command C
Machine Code to Execute Command D
etc.
Source Code File
A
B
C
D
E
F
G
H
Various 'commands' making up users source program

Figure 1: The interpretative cycle of common types of languages such as BASIC. After examining each command in the source file, the interpreter branches to the corresponding block of machine code; thus, program execution always remains within the interpreter.

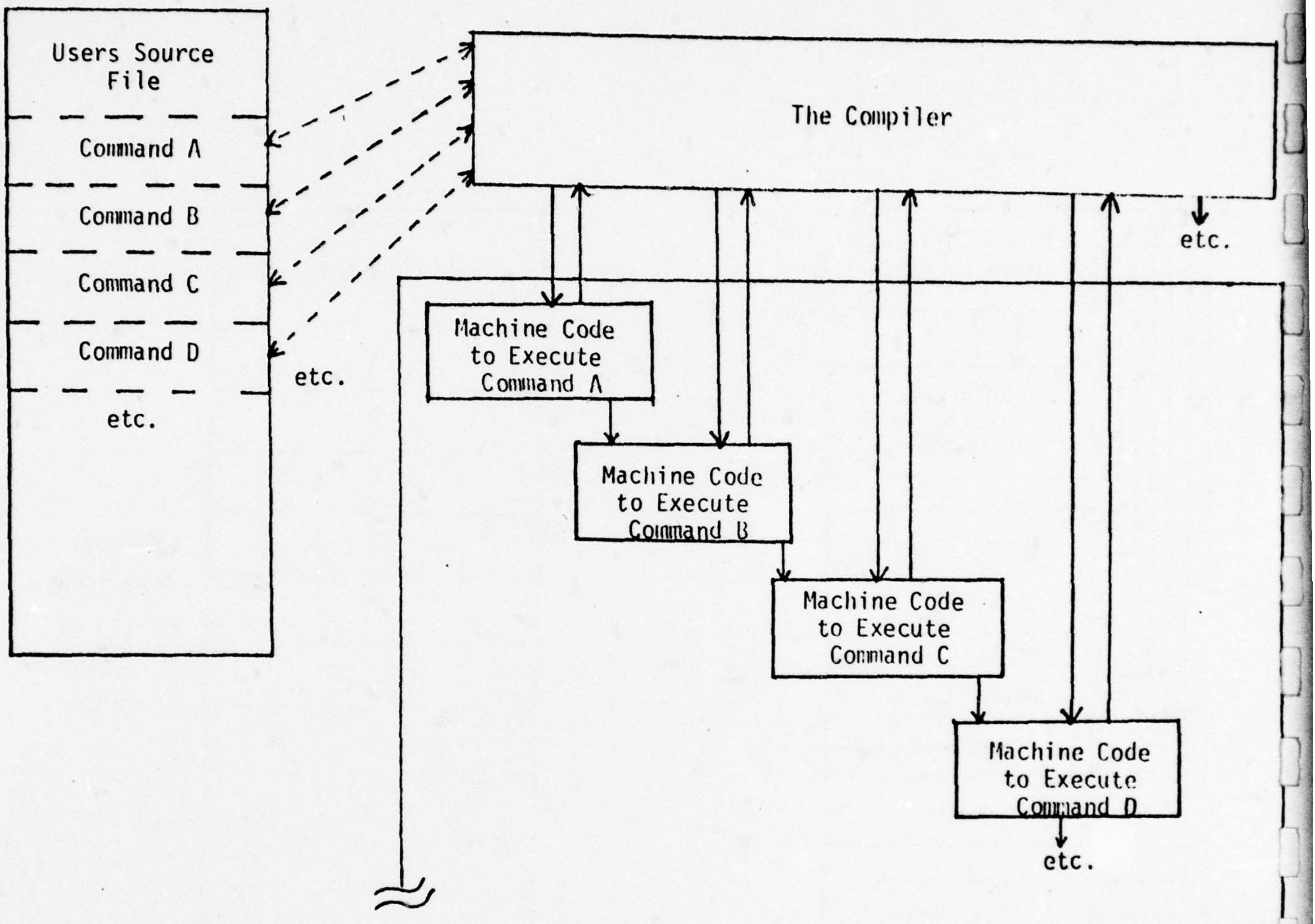

Users Source File
Command A
Command B
Command C
Command D
etc.
etc.
The Compiler
etc.
Machine Code to Execute Command A
Machine Code to Execute Command B
Machine Code to Execute Command C
Machine Code to Execute Command D
etc.

Figure 2. Note that the compiler transforms each source "command" into executable machine code. This code will be loaded and executed independently of the compiler.

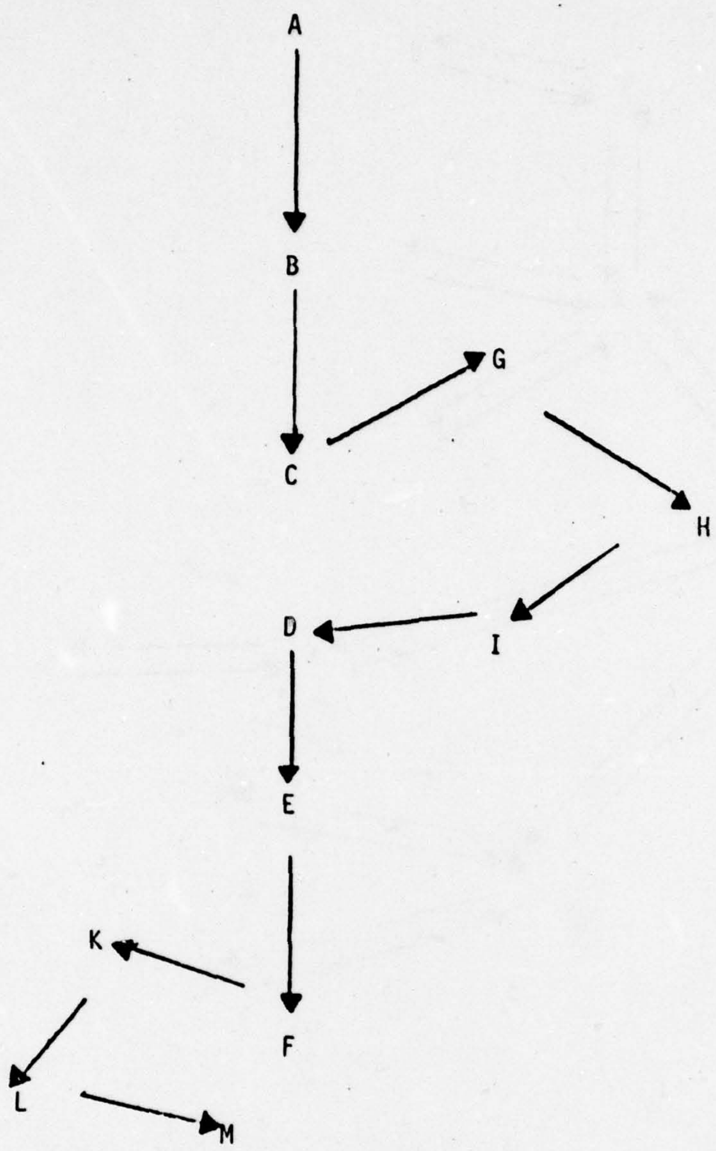


Figure 3. Programming structure using typical high-level languages. The flow of logic is linear, with an occassional branch or loop.

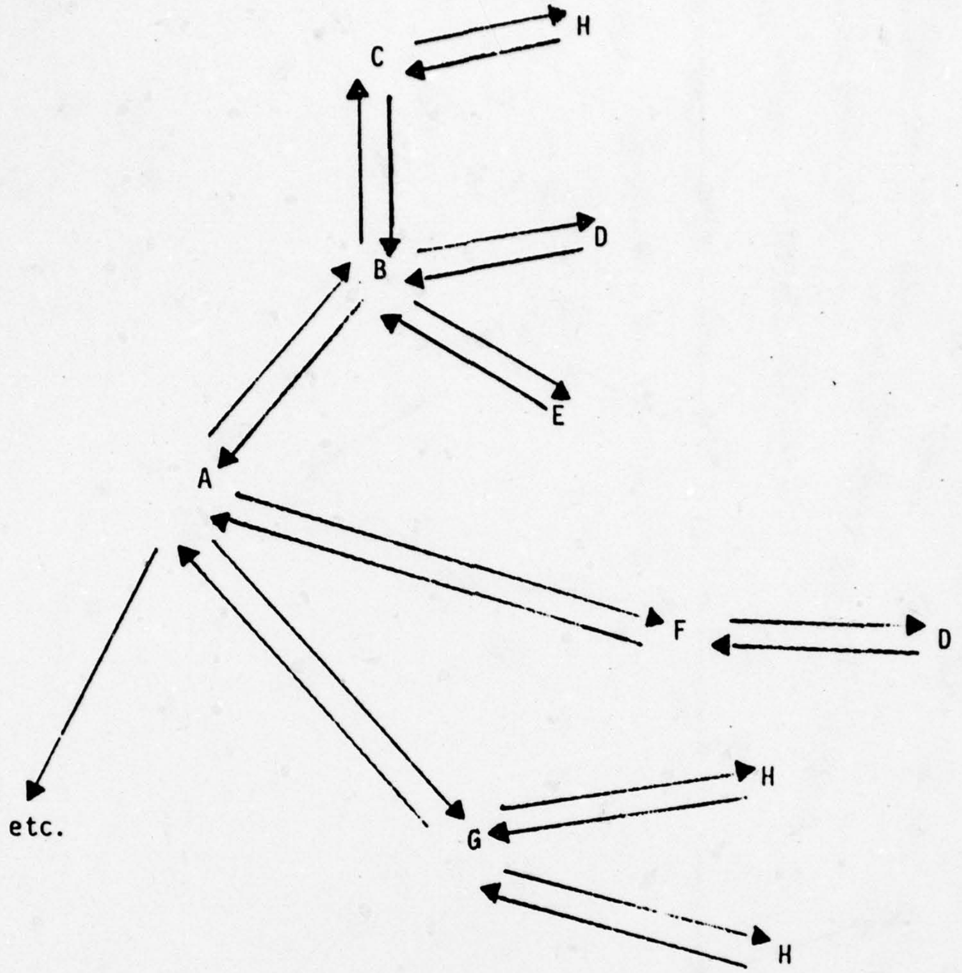


Figure 4. The 'threaded' code approach used in CONVERS. Note that the flow of logic threads its way in a very non-linear fashion.

CONVERS - an Interpretive Compiler

Part II

by

Scott B. Tilden and M. Bonner Denton

Department of Chemistry
University of Arizona
Tucson, Arizona 85721

INTRODUCTION

Part I of this series gave an introduction to a new type of software system called CONVERS. Several of the concepts behind CONVERS, including its ability to separate in time the compile and execute states, as well as its use of a stack and threaded code dictionary were discussed. This article will deal with a variety of programming examples using CONVERS designed to demonstrate how a conceptually advanced programming system can significantly decrease programming time and effort.

Manipulating the Stack

As stated in the previous article, the stack is an extremely important and useful structure offered by CONVERS. The stack is a first in-last out temporary storage location. Many calculators use a stack to internally store data values that the calculator is to operate on. Thus, when one enters a number it is pushed on the stack. When another number is entered, it is pushed on top of the previous number, an operator such as the '+' key adds the numbers on the stack together and replaces the two numbers on the stack with the sum. Simultaneously, the sum is displayed in the calculator display panel, i.e. the display presents the top number on the stack.

In CONVERS, the stack is manipulated in an almost identical fashion. A number can be pushed on the stack by the operator from a terminal; if the interpreter recognizes this character string as a number, it is converted internally (into binary) and pushed on the stack. Thus, to push octal (or decimal) five onto the stack, the user types the character '5' followed by a space. All values are stored on the stack as 16-bit values; thus, in the 8080 system, numbers are stored on the stack as two bytes. To examine the contents of the <u>top</u> number on the stack, a period (.) is typed; the top number on the stack is converted according to the specified number base and displayed on the terminal. One can specify base eight or ten by typing 'OCTAL' or 'DECIMAL', respectively, onto the terminal. Until the base is changed, all number I/O conversions will take place according to the current base value.

To minimize the capabilities of the stack, a variety of stack operators exist. Table A lists the most common operators along with a brief discussion of the function of the operator.

Table A is in no way complete; however, it does give some indication of the types of stack manipulation routines available in CONVERS. As with all CONVERS entries, the stack operators can be compiled into new entries. For example, suppose the user needs to add some constant, say 60, to a series of numbers on the stack and output the value to the terminal while maintaining the number on the stack. The user might first define the following entry, named '.SIXTY+':

```
: .SIXTY+  60  +  DUP  .  ;
```

The entry '.SIXTY+' illustrates several points. First, it is good programming practice to choose a name for a new entry so that the required function is mnemonically indicated. This practice will make it easier for the user to remember an entry's function at a later date. Second, the use is not limited to the operators supplied with the CONVERS package. The user can 'tailor-make' entries at will to suit the particular needs at hand. The third point to be made is that constants can appear inside of definitions as is the case with the '.SIXTY+' entry. When '.SIXTY+' is executed, the constant 60 is pushed on the stack, whereupon it is added to the number found on the stack originally. This number is duplicated and output. Once '.SIXTY+' has been compiled, it can itself be used in new entries. For example, the entry 'ANSWER', defined below, will output three numbers using '.SIXTY+' separated by one space between each number. The definition will also do a final carriage return-line feed.

```
: ANSWER  .SIXTY+  SPACE  .SIXTY+  SPACE  .SIXTY+  CRLF  ;
```

Obviously, this same format can also be obtained using a "DO-LOOP" which will be described shortly.

Constants and Variables

Every computer user recognizes the need to be able to define constants and variables. 'Local' constants are easily defined as was the case in the entry '.SIXTY+', where the constant with the value '60' resides in the entry itself. (A discussion of how a constant is stored in an entry must come later.) Many constants are, or can be, local in nature. The way the constant '60' is compiled into the entry '.SIXTY+' is an acceptable way of introducing local constants. However, some constants are not local in nature; i.e. the constant affects several aspects of the program which are not necessarily related in time or space. In this case, we must have the capability of defining 'global' constants. This means that the value of the constant must be easily accessible at any point during the execution of the 'program'. In CONVERS, one defines constants with the CONSTANT entry:

```
0 CONSTANT ZERO
```

In the above example, an entry 'ZERO' has been defined, initialized to the value zero. Whenever 'ZERO' is executed, the value zero will be pushed on the stack. Thus, any future entry can be compiled using 'ZERO' knowing the 'ZERO' will always push a zero on the stack when executed. Since any entry can access 'ZERO', 'ZERO' is a global constant.

Variable defining capabilities are as important as constants in any software system. Variables allow for dynamic changes in program execution to take place by reinitializing the variable to some other value. To create a variable in CONVERS, the 'VARIABLE' entry is used:

```
10 VARIABLE TIME
```

In this example, the entry 'TIME' has been initialized with the value ten. Whenever TIME is executed, the address where the variable value is stored is pushed on the stack. To get the value itself, the entry '@' is executed, '@' replaced the address found on the stack with the value stored at that address. Thus, by executing the character string 'TIME @', the value 10 will be pushed on the stack.

To change the value of a variable, the entry '!' is used. This entry stores the value at the second location under the stack at the address found on top of the stack. Thus, to reinitialize 'TIME' to the value '20', the following character string is executed:

```
20 TIME !
```

This string first pushes 20 and then the address of TIME onto the stack. Next, the top number on the stack is used as the address at which the second number is deposited. Whenever 'TIME @' is now executed, 20 will be pushed on the stack until reinitialized at a later date.

INDEVICE and OUTDEVICE

The 'INDEVICE' and 'OUTDEVICE' entries are provided as generalized I/O definitions; i.e. they will input or output data to or from the stack respectively, given the port address found on top of the stack. For example, the character string '0 10 OUTDEVICE' will output the value zero to port ten.. The 'INDEVICE' entry is used to input data and push this data on the stack. Again, the port address must also be on the stack before 'INDEVICE' is executed. For example, '10 INDEVICE' will input the value at port 10 and push it on the stack.

In a typical hardware configuration, there might be several I/O ports controlling a variety of external hardware. As an example, Figure 1 shows an example where a monitor and a D/A converter are 'tied' to two output ports, 30 and 31, respectively. An A/D converter and a high speed type reader are input devices, tied to input ports 40 and 41, respectively. However, most I/O devices also need encode (start) pulses as well as a means for testing flag status. Therefore, two other ports are needed. Input port 51 will serve as the flag port and output port 5Ø will be defined as the encode port. (Obviously, all port addresses are arbitrary.) Each bit of these ports will be tied to the corresponding encode or flag line of each device as shown in Figure 1.

Ignoring encode and flag functions, it is a simple matter to write I/O drivers to input or output data to these ports. For instance, a monitor 'driver' could be simply written as follows:

```
: MONITOR 3Ø OUTDEVICE ;
```

The definition 'MONITOR' when executed, would output the number on the stack to device 30, which is the monitor port. Similarly, other drivers could be written for each of the other devices; for instance, the 'A/D' definition below would input the data value from port 40 (the A/D port) and put this number on the stack whenever 'A/D' is executed:

```
: A/D 40 INDEVICE ;
```

It is a simple matter to include encode and flag testing software into these definitions; however, the reader must be familiar with the CONVERS conditional testing routines. Thus, a more thorough discussion of I/O software will come later in this article.

IF-ELSE-THEN (Conditional Testing)

The entries 'IF', 'ELSE' and 'THEN' are the CONVERS definitions for executing conditional branches. As an example, the definition '?DEVICE' will be defined as follows:

```
: ?DEVICE IF 77 MONITOR ELSE A/D THEN ;
```

The entry '?DEVICE', when executed, will test the top number on the stack for a non-zero condition. If this number is non-zero, the entries between 'IF' and 'ELSE' will execute and control will pass to whatever entries follow 'THEN'. Otherwise, the entries after 'ELSE' will execute. Given the previous definitions of 'MONITOR' and 'A/D', the entry '?DEVICE' would print a question mark (ASCII 77) on the monitor if the number on the stack was non-zero; otherwise, if the number was zero, the value of the A/D converter would be read and pushed on the stack.

Looping Structures

Conditional looping structures in CONVERS work much like those in other programming languages. For example, the 'VALUE' definition below will be defined to execute the previously defined entry 'A/D' one hundred times:

```
: VALUE 100 1 DO A/D LOOP ;
```

The DO-LOOP structure requires the upper and lower indices (in that order) to be on the stack before the loop is executed. Where these parameters come from is not important as long as they are there before execution of the loop. In the 'VALUE' example above the indices come from the definition itself (100 and 1); VALUE could just as well have been compiled as follows:

```
: VALUE DO A/D LOOP ;
```

In this example, the user (or some other definition) must supply the indices before 'VALUE' is executed. Obviously, leaving indices out of a looping definition makes the definition more general.

The special definition 'I' accesses the current lower index of a DO-LOOP during execution of the loop. As an example, the 'MONITOR-TEST' test program below will display the first 30 ASCII characters on the monitor:

```
: MONITOR-TEST 3Ø 1 DO I MONITOR LOOP ;
```

The definition 'I' would initially push the number 'one' on the stack the first time through the loop. This number is then sent to the monitor. The next time through the loop 'I' would push the number 'two' on the stack, and so on. The last time through the loop, 'I' would push on the stack the number thirty which is the upper index value.

CONVERS allows for the nesting of loops with the same general considerations as encountered in other common computer languages. The most important consideration is that an inner loop must lie entirely within the next outer loop. The special definition 'J' allows an inner loop to access the next outer loop's index:

```
: INNER-DISPLAY 2Ø Ø DO I J OUTDEVICE LOOP ;

  : INITIALIZE 1Ø Ø DO INNER-DISPLAY LOOP ;
```

The entry 'INITIALIZE' would, if executed, output the first 21 integer numbers to the first eleven I/O ports. Note that 'INNER-DISPLAY' gets the port address by executing 'J' which puts the outer index on the stack. The current inner index value (found by executing 'I') is sent to this port each time through the loop. If it was necessary to loop entries three levels deep, the 'K' definition would be used to access the current outer loop index.

BEGIN-HERE and BEGIN (Unconditional Looping)

The 'DO-LOOP' construct is useful in looping structures which require a known number of looping 'passes' to execute before the loop ends. In many cases, it is not known how many passes the loop will take before the loop should terminate; this is especially the case when a program loops until some specified condition (flag) is detected. As an example, the software driver controlling the high-speed reader will be defined below. It will be assumed that every 'read' operation from the reader port (port 41 as defined in Figure 1) will clear the flag:

```
: READER BEGIN-HERE 51 INDEVICE 1Ø AND
  IF 41 INDEVICE ELSE BEGIN THEN ;
```

Note that the 'READER' definition starts with the 'BEGIN-HERE' entry. This definition sets up a pointer to which the 'BEGIN' entry will refer back. The flag port contents are input; this value is logically 'ANDED' with the value 10. This octal value corresponds to the value of the flag bit from the reader. The logical 'AND' operation will leave either a zero or a 1Ø on the stack depending on the condition of the reader flag. A non-zero condition indicates that a new data value from the reader is ready for input; this value is read from port 41 and the 'READER' definition ends. A zero flag will cause the 'BEGIN' entry to execute, sending control back to the 'BEGIN-HERE' pointer. Thus, this structure causes the entry to continually loop until the respective flag has been set. Similarly, the A/D driver can be extended to include flag checking:

```
: A/D BEGIN-HERE 51 INDEVICE 4 AND
  IF 4Ø INDEVICE ELSE BEGIN THEN ;
```

Note that the flag bit (bit 2) corresponds to an octal value of four. The 'A/D' definition will continually loop until bit 2 has been set, at which time the data from the A/D will be read from port 4Ø.

Software drivers can, likewise, be written to handle the rest of the I/O devices in Figure 1. Obviously, it would be an easy matter to write a software driver for any user-defined I/O device given any hardware configuration. As is the case with all CONVERS entries, these drivers are compiled when originally defined, retaining a speed advantage in the sometimes critical execution-time I/O environment.

CONCLUSION

Although many of the standard high-level CONVERS definitions have not been covered, this article should acquaint the user with the types and nature of the CONVERS entries already defined. Most programming needs can draw upon these standard entries. It is the user's responsibility to compile the required specialized routines from these standard entries for the particular application at hand.

This work was supported in part by the Office of Naval Research and an Alfred P. Sloan Research Fellowship to M. Bonner Denton.

TABLE A

OPERATOR	FUNCTION
.	Converts and outputs the top number on the stack according to the user-selected numerical base.
DUP	Duplicates the top number on the stack and pushes the copy on top of the stack.
SWAP	Swaps the top two numbers on the stack.
+	Adds top two numbers on the stack--replaces two numbers with the sum.
-	Subtracts the top number from the bottom number and replaces the two numbers with the difference.
MIN	Replaces the top two numbers on the stack with the minimum of the two numbers.
MAX	Replaces the top two numbers on the stack with the maximum of the two numbers.
X UNDER	Locates the number at the location (specified by displacement 'X' from the top of the stack) and copies it on top of the stack. Thus, to COPY the fourth number under the stack back on top of the stack, one would type '4 UNDER' on the terminal.
SWITCH	'SWITCHES' the 8 LSB's and 8 MSB's of the number on top of the stack.
DROP	Remnves the top number on the stack.
POP	'POPS' the top byte on the stack and puts it into the 'A' register.

PUSH	'PUSHES' the byte in the 'A' register to the top of the stack.
<	Replaces the top two numbers on the stack with a '1' if the bottom number is less than or equal to the top number; otherwise, a zero is pushed on the stack.
>	Replaces the top two numbers on the stack with a '1' if the bottom number is greater than or equal to the top number; otherwise, a zero is pushed on the stack.
=	Replaces the top two numbers with a '1' if the two numbers are equal; otherwise, a zero is pushed on the stack.
0<	Replaces the top number on the stack with a '1' if the number is negative (MSB is one); otherwise, a zero is pushed on the stack.
COMPLEMENT	Complements the top number on the stack (one's complement).
MINUS	Complements and increments the top number on the stack by one (two's complement).
AND	Logically 'AND's' top two numbers on the stack, the result replaces the two numbers.
1+	Adds one to the top number on the stack. ('1+' operates much faster than adding one to the number on stack.)
1-	Decrements top number on the stack by one. (This is a fast decrement.)

TABLE A (Cont.)

A sample of the stack operators provided in the Standard High-Level Dictionary.

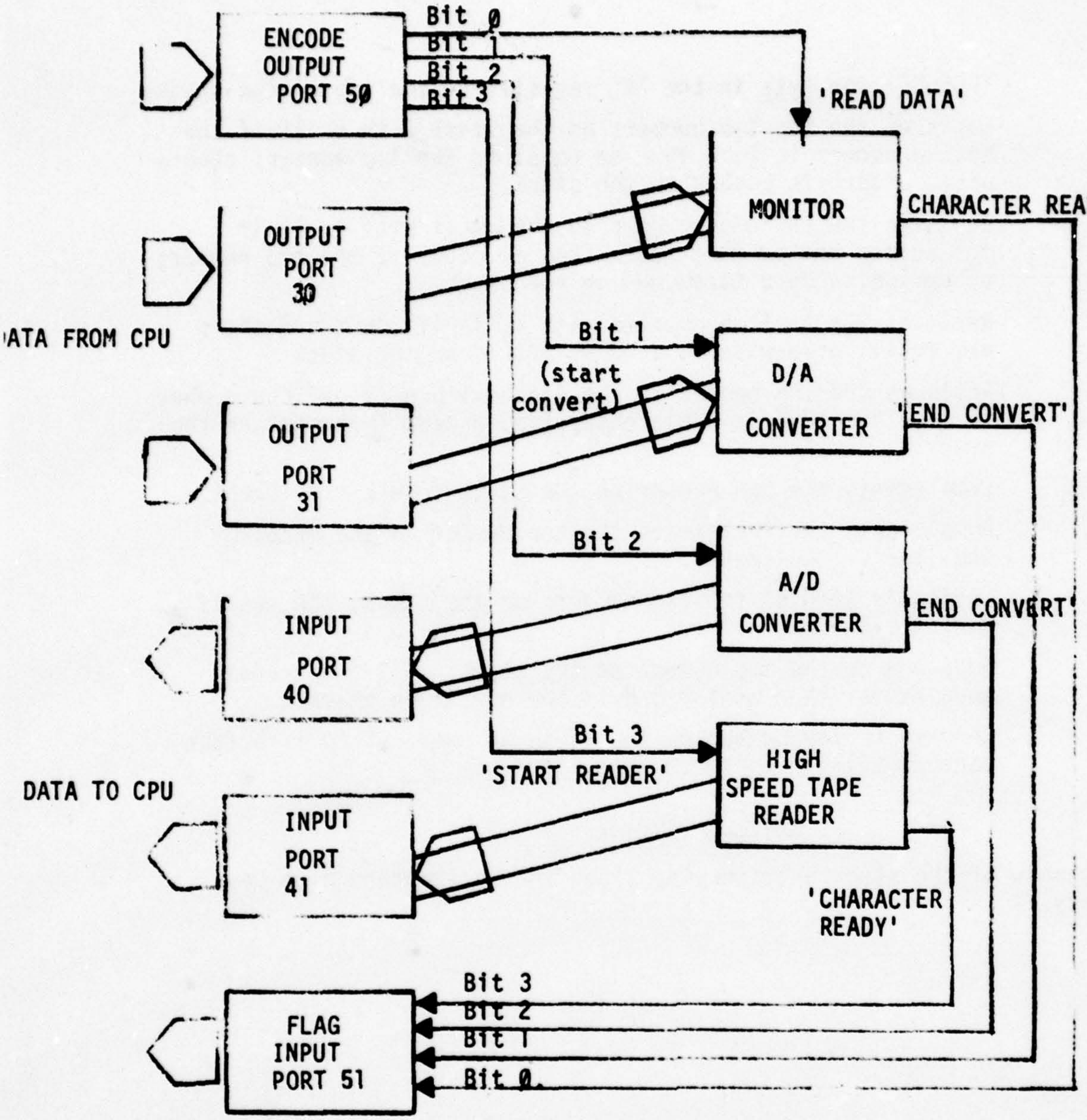


Figure 1. Each port is tied to an I/O device. Port 50 and 51 serve as 'flag' and 'encode' ports, respectively. One bit of the latter ports is tied to the corresponding I/O devices.

CONVERS - an Interpretive Compiler

Part III

by

Scott B. Tilden and M. Bonner Denton

Department of Chemistry
University of Arizona
Tucson, Arizona 85721

INTRODUCTION

The first two articles in this series discussed the fundamental concepts, advantages, and applications of CONVERS. CONVERS is relatively easy to understand at all levels, making it useful for those experimenters who want to both customize it for unique environments as well as to write new versions for other CPU's. Given a source listing of CONVERS one can 'decode' the system. However, by understanding the basic principles of CONVERS, decoding and translating efforts are vastly simplified. The aim of this article is to describe these basic principles and to give examples of these principles by describing particular sections of the code. This article will conclude with an 8080 object code listing of the CONVERS initial machine code dictionary (ISCD) and a source code listing of the CONVERS standard high-level dictionary; an object lising is not necessary.

The Dictionary

The most basic concept of CONVERS is the dictionary which was described in a general manner in Parts I and II. The dictionary is made up of a series of entries, one stacked on top of the other. Each entry has four elements associated with it: the name, the link, the code pointer, and the body of the entry containing executable machine code. Each of these elements will be described in turn.

The name of the entry is defined in the first four bytes of the entry. The first byte contains the total number of characters of the entry name (see Figure 1). Following the character count are three bytes which contain the first three characters of the name (if the name has less than three characters, the extra bytes will be zeroed). All entries are uniquely defined by these four parameters. This is important to keep in mind since an entry called 'DUMB' would have the same CONVERS name as a 'DUMP' entry. Since a potential problem could be created by different entries with identical names, CONVERS searches the dictionary for an identical name whenever the user creates a new entry. The user is warned of the existence of the duplicate entry by the diagnostic message, 'SURE', being displayed on the terminal. The user has the option of redefining the entry by typing a carriage return character and entering another name.

The link of each entry is contained in the next two bytes after the name. The link always points to the first byte (the character count) of the entry immediately below it in the dictionary. The very first entry has a link word of zero. Given the linking structure of the dictionary, it is easy to visualize how dictionary searches are accomplished. For instance, suppose the user wishes to execute the period (.) entry. Remember from the previous article that this entry converts and outputs the top number on the stack. The user would type the period followed by a space. The 'SEARCH' entry is called by the main executive routine. The entry 'SEARCH' compares the period against the last entry in the dictionary. If this entry does not match, 'SEARCH' compares the next entry in the dictionary. Remember that the location of the next entry is pointed to by the link of the previous entry (see Figure 2). Searching stops whenever the entry is located or a link word of zero is detected.

The code pointer follows the link word. It points to the starting location of the entry; this is normally the first byte in the body. By changing the code pointer address, any entry can begin execution at any point in memory. This is what in fact is done when 'OCTAL' or 'DECIMAL' is typed on the terminal; the code pointer of the corresponding number conversion routines are changed to point to the respective octal or decimal version. The 'EXECUTIVE' routine compiles or executes entries at the address found at the code pointer.

The body of each entry contains executable machine code. This code is supplied in four basic ways. The first is for the user to compile machine code definitions; the code placed in the body of the definition is that explicitly supplied by the user. The second way is for other definitions to place code in the body of the new definition (i.e. assembler routines). The third is to compile a new entry using previous entries; in this case, the 'EXECUTIVE' supplies 'calls' to the entries defining the new definition. The destination address of the 'call' is the address found at the code pointer of the entry. In the fourth case, machine code is supplied by the 'NUMBER' routine which deposits a call to 'LITERAL' followed by the number found on top of the stack. The 'number handling' routines will be discussed later.

The Stack

As outlined in the first article, the dictionary resides in low memory and the stack in high memory. The two routines, 'PUSH' and 'POP', add and remove single byte quantities from the stack while respectively decrementing and incrementing the stack pointer. The stack pointer always points to the last datum to be added to the stack. These two routines also check for stack underflow and overflow. Flow charts describing these routines are given in Figures 3 and 4.

Since all stack operations are ultimately performed by the 'PUSH' and 'POP' entires, double byte stack manipulation routines must execute 'PUSH' or 'POP' twice in order to add or remove two byte values. Most stack entries treat numbers as 16 bit values.

Simple number manipulations, including logical operations, are performed in the internal registers of the CPU. The two routines, 'STKDE' and 'STK-BC', allow two byte values on the stack to be deposited into these registers through manipulation of the 'POP' entry. Since the 'POP' entry only affects the H & L register pair, data may be deposited into either the 'B & C' or 'D & E' register pair without affecting the contents of the other register pair. Stack operators that perform mathematical or logical functions call either or both the 'STK-BC' or 'STKDE' entries, perform the function by proper manipulation of these internal registers and push the result back on the stack. This latter operation is performed by calling either the 'BC-STK' or 'DE-STK' entry. These entries push the respective register contents back on the stack by manipulation of the 'PUSH' routine. Since the 'PUSH' routine only affects the contents of the 'H & L' register pair, the 'BC-STK' or 'DE-STK' routine will also execute without affecting the contents of the other register pair.

The Executive

The most important routine in CONVERS is the 'EXECUTIVE' routine. The 'EXECUTIVE' routine is the essence of CONVERS providing the system monitor, execute and compile loop, and general control of the system. It is the function of the 'EXECUTIVE' to pull together all the other routines in the initial machine code dictionary to make a working system.

Upon completion of any entry entered on the terminal, control will always be passed back to the EXECUTIVE. The EXECUTIVE initially pushes its starting address on the hardware return stack (the 8080 return stack is a CPU controlled stack that contains, primarily, the return address of one routine when executing a 'call' to another and should not be confused with the main stack in high memory). Since all entries contain a return instruction, upon execution of the return instruction, control will be passed back to the start of the 'EXECUTIVE' providing that the return stack has not been perturbed. Due to the difficulty of mixing data and return addresses on one stack, a software stack was created to hold data and other program parameters. The two entries, 'PUSH' and 'POP', have the responsibility of controlling the software stack.

Since all entries to be executed or compiled come from a terminal or a mass storage device, the 'EXECUTIVE' first calls the 'UPDICT' routine. This routine has the responsibility of depositing the entry name at the end of the dictionary. A flow chart describing the 'UPDICT' routine is shown in Figure 5. A system variable, the Dictionary Pointer (D.P.), always contains the last byte to be permanently added to the dictionary. The 'UPDICT' routine deposits the entry name starting after the dictionary pointer.

The 'EXECUTIVE' must now call the 'SEARCH' entry to determine whether the entry entered from the terminal (or mass storage) is contained in the dictionary. The 'SEARCH' routine, upon completion, either leaves a zero or the address of the code pointer of the located entry on the stack. A non-zero value on the stack indicates that the entry name deposited by 'UPDICT' matches an entry name in the dictionary. At this point, the EXECUTIVE will branch to execute one of two separate functions depending on the value of the number on the stack. If this number is zero, the 'EXECUTIVE' will branch to the 'NUMBER' routine. This particular branch will be described in detail later.

If the value left on the stack by the 'SEARCH' entry is non-zero, the 'EXECUTIVE' must determine whether the located entry is to be compiled or executed. Since some entries must be executed during a compiling procedure while others need to be located and the starting addresses compiled, a mechanism must be provided to designate the proper action. The 'EXECUTIVE' determines this by comparing the 'precedence' value of the entry against a 'STATE' variable. The precedence value of all entries is contained in the most significant bit of the character count byte of each entry (see Figure 1). The precedence value of each entry is either zero or one depending on whether the entry has low or high precedence. If the precedence value is equal to or greater than 'STATE', the entry is executed. Otherwise, the entry is compiled.

The 'STATE' variable is controlled by the colon (:) and semicolon (;) entries. The colon entry sets the 'STATE' to one. Only the entries with precedence values of one will execute, all other entries will, instead, be compiled. As an example, suppose the system is in the compile mode ('STATE' equal to one) and the user types 'BELL' on the terminal. Since the precedence values of definitions such as 'BELL' are zero, these entries are compiled; i.e. a 'call' to the address of 'BELL' is deposited at the end of the dicitonary and the dictionary pointer is incremented. However, if the system is in the execute mode, 'STATE' will be zero and 'BELL' will ring the terminal bell. The colon entry also has the function of supervising the input of the new dictionary entry name.

The semicolon (;) entry is used to end the compilation of the new entry. It does this by zeroing 'STATE'. This has the effect of returning the system back to the execute mode as discussed above. The semicolon entry also deposits a return instruction (311 octal) into the dictionary and officially adds the new entry to the dictionary by updating the 'LINKWORD' variable. The 'LINKWORD' variable always points to the first byte (the character count) of the last entry to be compiled into the dictionary. This variable is used by the 'SEARCH' entry as the location to begin dictionary searches. The 'LINKWORD' variable is not updated until the entry has been totally compiled. This explains why a new entry can compile into itself an 'older' entry with the same dictionary name. A flow chart describing the 'EXECUTIVE' routine is shown in Figure 6.

The Number Routine

In the previous discussion of the 'EXECUTIVE' it was stated that if an 'entry' entered on the terminal was not located in the dictionary, a jump to the 'NUMBER' routine was made. The logical reason for jumping to 'NUMBER' at this point is simple--if the 'entry' is not a definition, it must either be interpreted as a number or as an error. 'NUMBER' determines in a very simple fashion whether the 'entry' entered on the terminal (or from mass storage) should be interpreted as a number; it tests each character to see if it is between octal 60 and 67 inclusive for the octal case and 60 and 71 inclusive for the decimal case. Obviously, these ASCII values represent the numbers allowed in each respective number system. If one or more characters entered lie outside the above values, an error is assumed and the error message '?2' is output to the terminal. One could have misspelled a definition name or thought that a certain definition existed which did not at that time (see Figure 7).

Suppose the character string can be interpreted as a number. In this case, the character string is converted to a 16-bit binary value depending on the base chosen (octal or decimal). This is done by a call to the 'OCTAL' or 'DECIMAL' conversion routine, respectively. The converted binary number is then 'pushed' on the stack as two bytes.

The 'NUMBER' routine must now do one more test before returning. It must test 'STATE' to see whether the number should be left on the stack ('STATE' equal to zero) or compile the number into the dictionary if 'STATE' is one. It does the latter by compiling a call to 'LITERAL' and then depositing the number into the dictionary, as discussed in the previous article (remember the '.sixty+' compilation). The final return instruction of 'NUMBER' places control back to the start of the 'EXECUTIVE'.

CONCLUSION

This article has focussed on an in-depth look into the logical program flow of the CONVERS Initial Machine Code dictionary (IMCD). Although a number of routines are needed to successfully emulate the Initial Machine Code dictionary (IMCD) on other CPU's, it should be emphasized that each particular routine is logically rather simple. It is only when taken as a whole that the CONVERS system becomes so powerful.

An object listing of the Initial Machine Code dictionary (IMCD) is given in Figure 8. An octal loader should be used to load memory starting at octal 100. The Initial Machine Code dictionary's starting address is 1752 octal. Once it is loaded and running, it will itself compile the rest of Standard CONVERS; a source listing is given as Figure 9. The version given here is for 12k of memory. To reconfigure for another memory size, the procedure outlined in Table I should be followed.

Table II lists the contents and memory location of the terminal I/O routines. These routines may also need to be reconfigured (or rewritten) depending on the user's particular terminal or interface.

This work was partially supported by the Office of Naval Research and by an Alfred P. Sloan Foundation Research Fellowship to M. Bonner Denton.

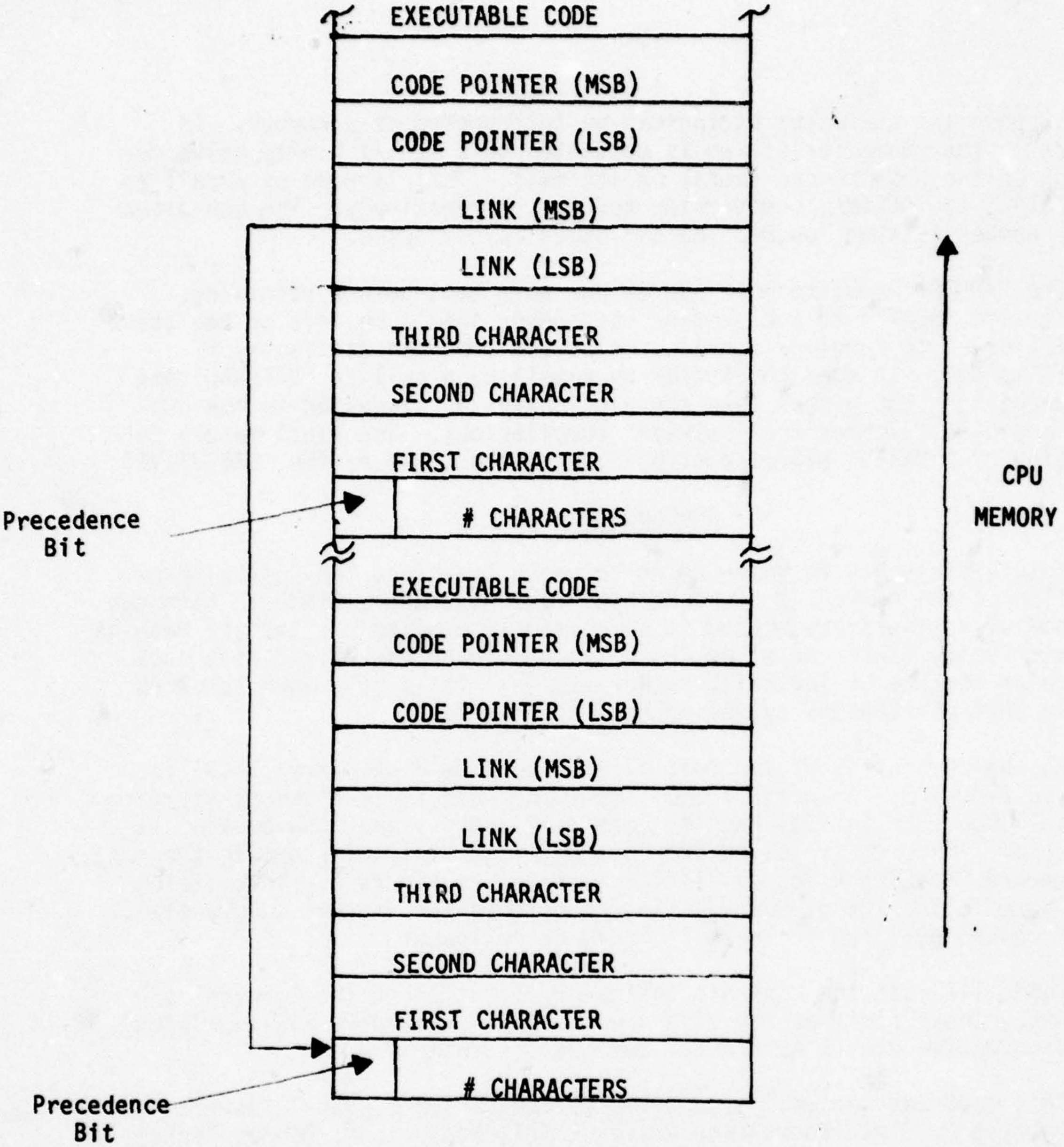


Figure 1. Each entry in the dictionary is named by the above format.

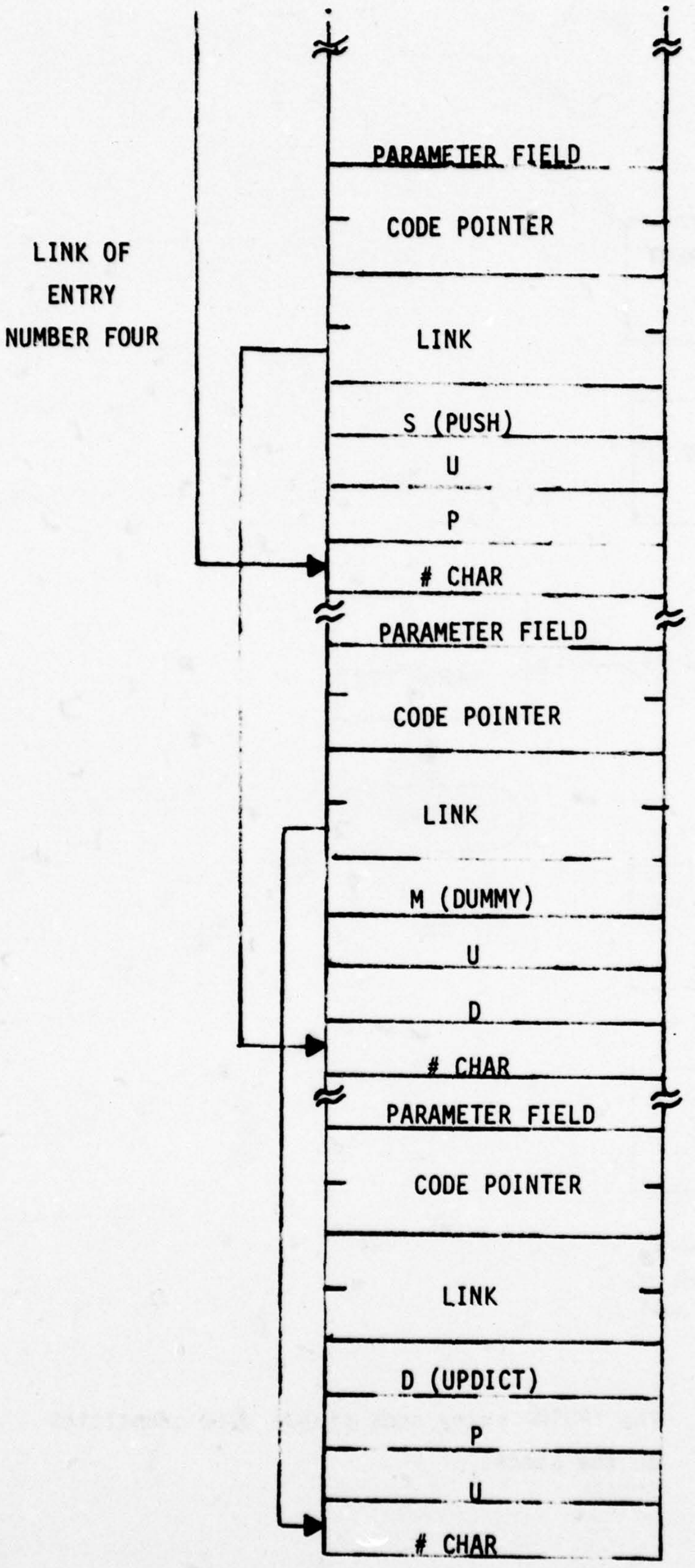


Figure 2. Note the linking nature of the dictionary. Each link points to the first memory location of the previous entry in the

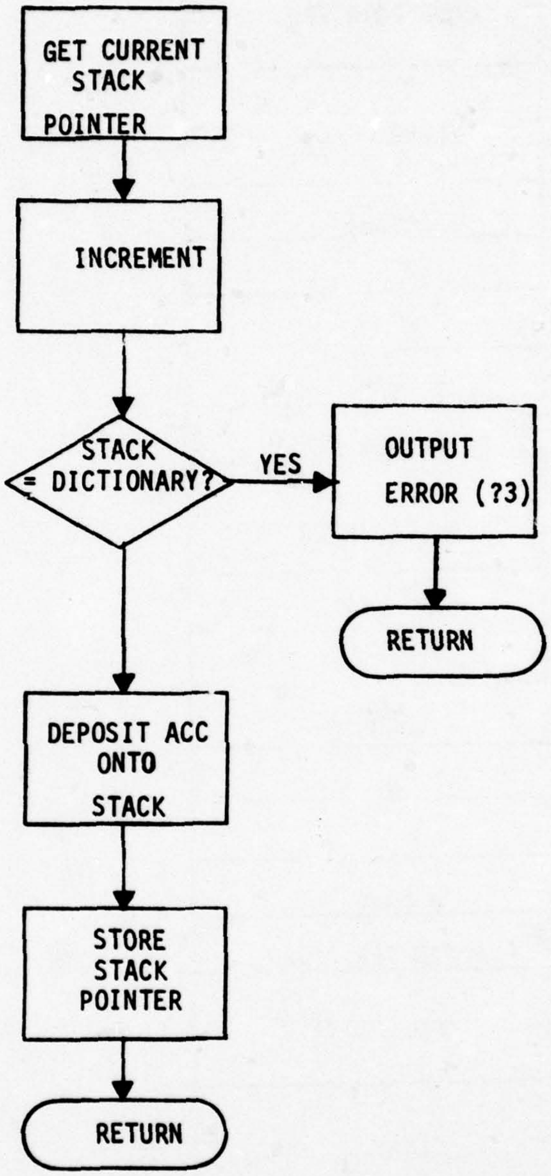


Figure 3. The 'PUSH' entry adds single byte quantities to the stack.

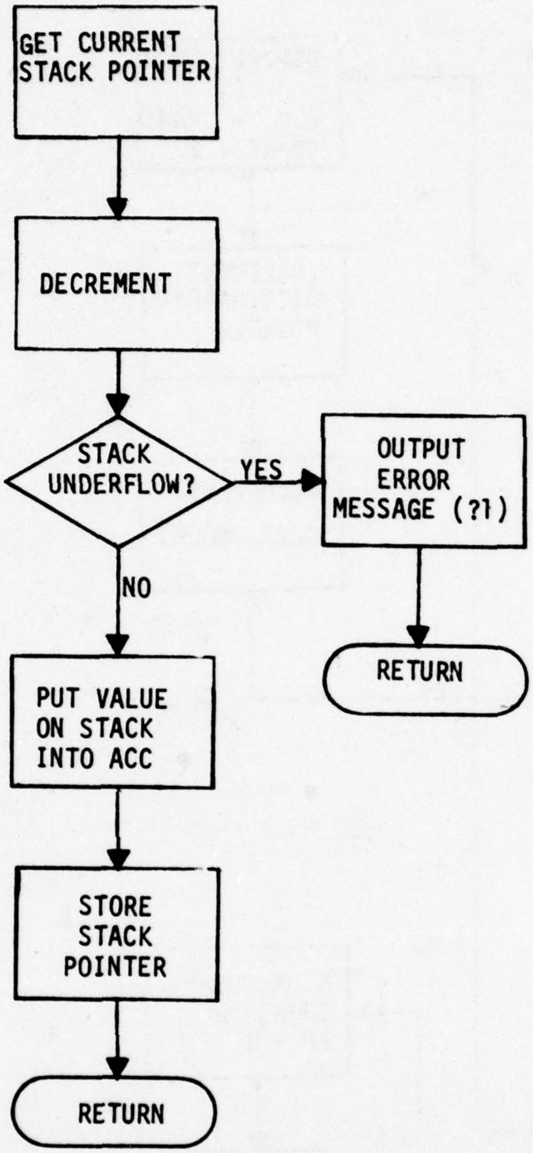

GET CURRENT STACK POINTER
DECREMENT
STACK UNDERFLOW?
YES
NO
OUTPUT ERROR MESSAGE (?1)
RETURN
PUT VALUE ON STACK INTO ACC
STORE STACK POINTER
RETURN

Figure 4. The 'POP' entry removes single byte quantities from the stack to the accumulator.

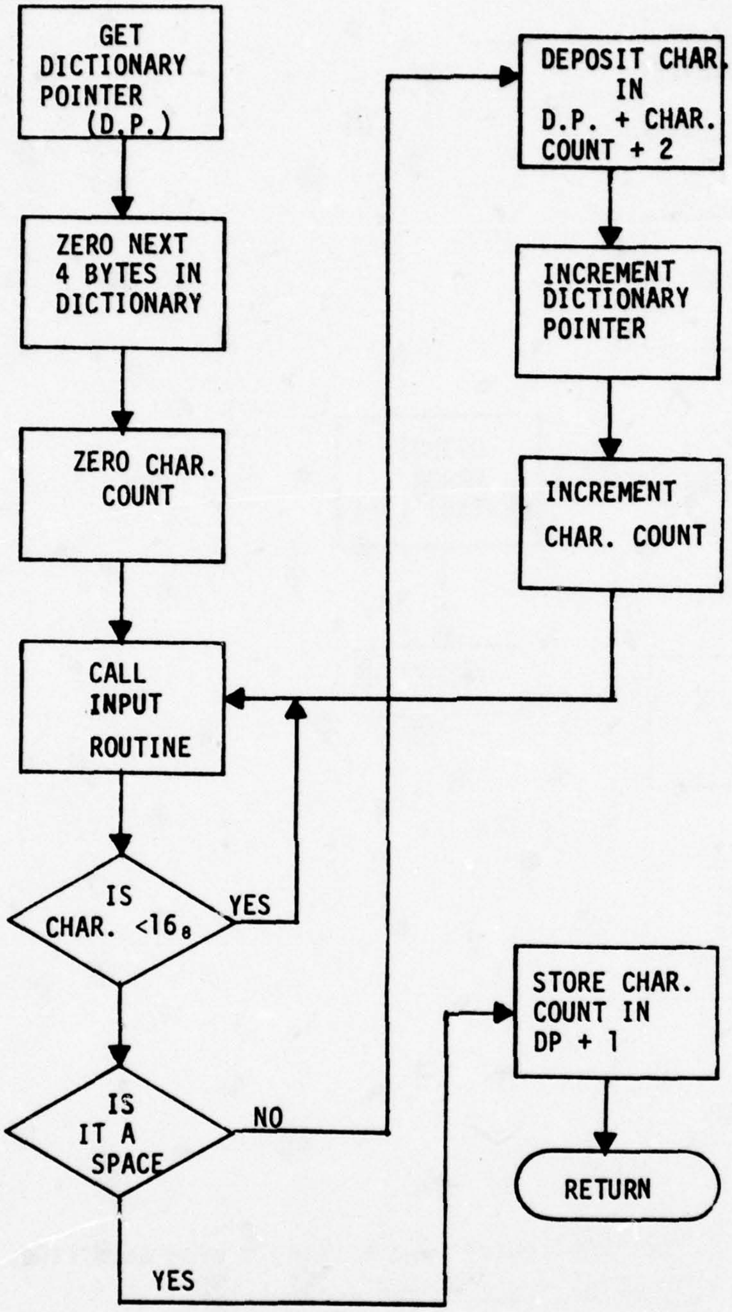

GET DICTIONARY POINTER (D.P.)
ZERO NEXT 4 BYTES IN DICTIONARY
ZERO CHAR. COUNT
CALL INPUT ROUTINE
IS CHAR. <16₈
YES
IS IT A SPACE
NO
YES
DEPOSIT CHAR. IN D.P. + CHAR. COUNT + 2
INCREMENT DICTIONARY POINTER
INCREMENT CHAR. COUNT
STORE CHAR. COUNT IN DP + 1
RETURN

Figure 5. The 'UPDICT' flow chart. 'UPDICT' is used to input dictionary names or numbers from the terminal or mass storage.

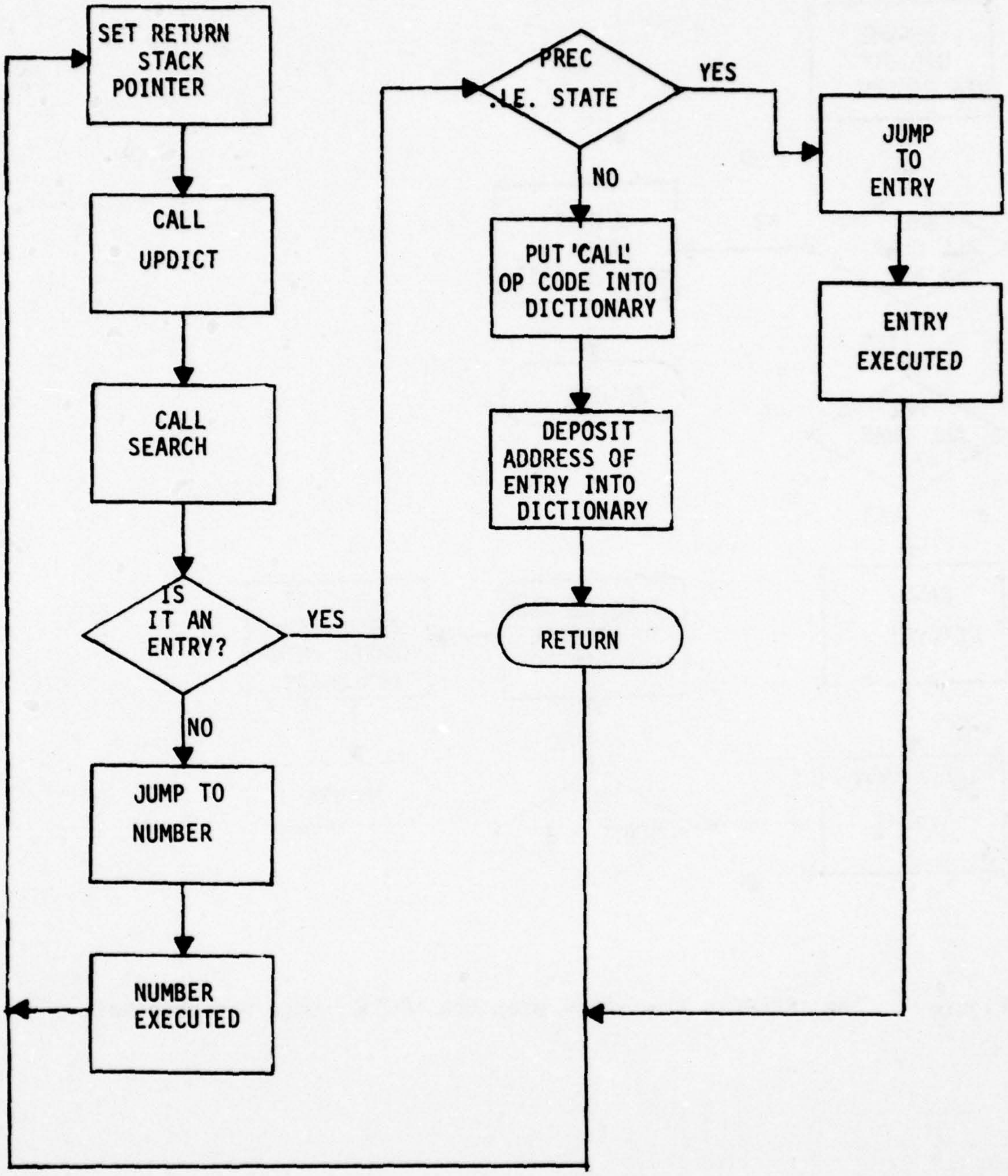


Figure 6. The flow chart of the 'EXECUTIVE' routine.

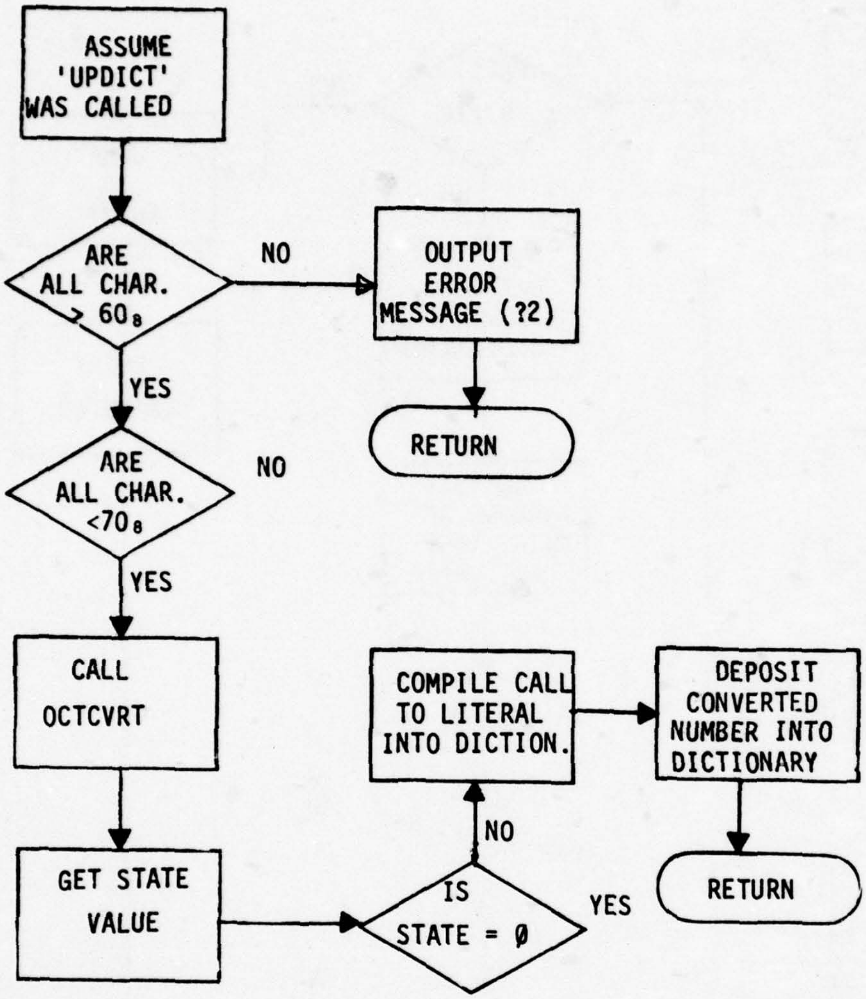

ASSUME 'UPDICT' WAS CALLED
ARE ALL CHAR. > 60₈
NO
OUTPUT ERROR MESSAGE (?2)
RETURN
YES
ARE ALL CHAR. <70₈
NO
YES
CALL OCTCVRT
GET STATE VALUE
IS STATE = Ø
NO
YES
COMPILE CALL TO LITERAL INTO DICTION.
DEPOSIT CONVERTED NUMBER INTO DICTIONARY
RETURN

Figure 7. The 'NUMBER' flow chart when the 'OCTAL' mode has been set.

CONVERS INITIAL MACHINE CODE DICTIONARY

PART 1

```
ØØØ ØØ6 367 ØØ4 ØØ6 125 12Ø 1Ø4 ØØØ ØØØ 114 ØØØ 257 Ø52 1ØØ ØØØ
 Ø43 167 Ø43 167 Ø43 167 Ø43 167 Ø53 Ø53 ØØØ 127 315 377 ØØ4 376
 Ø16 332 134 ØØØ 376 Ø4Ø 312 157 ØØØ Ø24 167 Ø43 3Ø3 134 ØØØ Ø52
 1ØØ ØØØ Ø43 162 311 365 325 Ø52 1ØØ ØØØ Ø21 Ø1Ø ØØØ Ø31 175 Ø57
 137 174 Ø57 127 Ø23 Ø52 3Ø7 ØØØ Ø31 322 217 ØØØ 321 361 311 ØØ6
 Ø63 Ø76 Ø77 315 Ø24 ØØ1 Ø76 Ø4Ø 315 Ø24 ØØ1 17Ø 315 Ø24 ØØ1 257
 Ø62 337 ØØ3 3Ø3 352 ØØ3 ØØ5 1Ø4 125 115 1Ø4 ØØØ 256 ØØØ 315 366
 ØØ4 311 ØØØ ØØ4 12Ø 125 123 246 ØØØ 273 ØØØ 315 165 ØØØ Ø52 3Ø7
 ØØØ Ø53 167 Ø42 337 ØØØ 311 ØØØ Ø6Ø ØØ3 12Ø 117 12Ø 263 ØØØ 321
 ØØ3 Ø52 3Ø7 ØØØ Ø51 174 Ø67 Ø77 Ø37 Ø74 376 Ø61 332 344 ØØØ ØØ6
 Ø61 333 221 ØØØ Ø52 3Ø7 ØØØ 176 Ø43 Ø42 3Ø7 ØØØ 311 ØØ6 124 124
 131 311 ØØØ 365 ØØØ 315 Ø5Ø ØØ1 315 2Ø1 ØØ1 Ø24 Ø12 315 Ø24 ØØ1
 ØØ3 Ø35 3Ø2 374 ØØØ Ø25 3Ø2 374 ØØØ 311 ØØØ ØØØ ØØ6 117 125 124
 355 ØØØ Ø24 Ø71 365 333 ØØØ 346 2ØØ 312 Ø25 ØØ1 361 323 ØØ1 311
 ØØ5 123 124 113 Ø14 ØØ1 Ø52 ØØ1 315 321 ØØØ 127 315 321 ØØØ 137
 311 ØØ1 1ØØ ØØØ ØØØ Ø4Ø ØØ1 Ø71 ØØ1 315 Ø52 ØØ1 Ø32 315 273 ØØØ
 Ø23 Ø32 315 273 ØØØ 311 ØØ2 Ø61 1ØØ ØØØ Ø61 ØØ1 116 ØØ1 315 Ø5Ø
 ØØ1 Ø32 315 273 ØØØ 311 ØØ3 1Ø4 125 12Ø 1Ø6 ØØ1 136 ØØ1 315 Ø5Ø
 ØØ1 315 16Ø ØØ1 315 16Ø ØØ1 311 ØØ6 1Ø4 1Ø5 Ø55 126 ØØ1 16Ø ØØ1
 173 315 273 ØØØ 172 315 273 ØØØ 311 ØØ6 123 124 113 15Ø ØØ1 2Ø1
 ØØ1 315 321 ØØØ 1Ø7 315 321 ØØØ 117 311 ØØ6 1Ø2 1Ø3 Ø55 171 ØØ1
 222 ØØ1 171 315 273 ØØØ 17Ø 315 273 ØØØ 311 ØØ4 123 127 1Ø1 212
 ØØ1 243 ØØ1 315 2Ø1 ØØ1 315 Ø5Ø ØØ1 315 222 ØØ1 315 16Ø ØØ1 311
 ØØ4 117 1Ø5 122 233 ØØ1 27Ø ØØ1 Ø52 1ØØ ØØØ Ø43 353 315 16Ø ØØ1
 311 ØØ4 11Ø 1Ø5 1Ø1 26Ø ØØ1 311 ØØ1 Ø52 1Ø2 ØØØ 353 315 16Ø ØØ1
 311 ØØ2 ØØØ ØØ2 Ø61 Ø53 ØØØ 3Ø1 ØØ1 333 ØØ1 315 Ø5Ø ØØ1 Ø23 315
 16Ø ØØ1 311 ØØ4 117 126 1Ø5 323 ØØ1 353 ØØ1 Ø76 Ø6Ø ØØØ 147 Ø56
 ØØØ Ø53 136 Ø53 126 315 16Ø ØØ1 311 ØØØ ØØØ ØØ4 1Ø4 122 117 343
 Ø71 ØØ3 ØØ2 315 321 ØØØ 315 321 ØØØ 311 ØØ6 123 1Ø5 1Ø1 373 ØØ1
 ØØ2 ØØ2 Ø52 122 ØØØ Ø42 135 ØØ2 353 Ø52 1ØØ ØØØ Ø43 Ø16 Ø24 Ø32
 346 177 276 3Ø2 Ø62 ØØ2 Ø43 Ø23 Ø15 3Ø2 Ø37 ØØ2 Ø23 Ø23 315 16Ø
 ØØ1 311 Ø52 135 ØØ2 353 Ø23 Ø23 Ø23 Ø23 Ø32 326 ØØØ 157 Ø23 Ø32
 312 121 ØØ2 ØØØ ØØØ 147 Ø42 135 ØØ2 353 Ø52 1ØØ ØØØ Ø43 3Ø3 Ø35
 ØØ2 326 ØØØ 3Ø2 1Ø3 ØØ2 147 353 3Ø3 Ø56 ØØ2 ØØØ Ø2Ø 311 ØØØ ØØ4
 1Ø3 122 114 Ø12 ØØ2 147 ØØ2 Ø76 Ø15 315 Ø24 ØØ1 Ø76 Ø12 315 Ø24
 ØØ1 311 ØØ5 123 12Ø 1Ø1 137 ØØ2 172 ØØ2 Ø76 Ø4Ø 315 Ø24 ØØ1 311
```

```
004 005 132 105 122 162 002 211 002 315 201 001 076 000 270 302
233 002 271 302 233 002 257 311 000 000 004 067 311 000 000 000
007 117 103 124 201 002 250 002 052 100 000 043 106 353 257 147
157 076 005 273 322 301 002 023 032 326 060 007 007 007 157 006
005 023 032 326 060 205 157 005 312 321 002 051 051 051 303 301
002 353 315 160 001 311 006 116 125 115 240 002 336 002 315 270
001 315 071 061 315 270 001 315 333 001 315 052 001 315 201 001
032 376 060 332 020 003 376 070 322 020 003 023 015 302 360 002
315 250 002 072 337 003 376 000 310 315 256 003 315 076 003 311
021 040 003 315 160 001 021 003 000 315 160 001 315 365 000 311
077 040 062 001 041 040 000 326 002 053 003 315 050 001 315 201
001 353 161 043 160 311 001 054 000 000 043 003 076 003 315 201
001 052 102 000 043 161 043 160 042 100 000 311 000 000 000 007
070 000 000 000 005 105 116 124 066 003 134 003 052 100 000 043
042 177 003 043 043 043 043 353 052 102 000 175 022 023 174 022
023 142 153 043 043 175 022 023 174 022 353 042 103 000 311 001
006 000 000 004 114 111 116 124 003 213 003 052 177 003 042 172
002 311 002 061 054 002 203 003 232 003 315 201 001 052 100 000
043 161 042 100 002 311 007 114 111 124 222 003 256 003 021 270
003 315 160 001 315 316 003 311 000 341 124 135 043 043 345 315
160 001 315 371 001 311 007 103 117 115 246 003 316 003 076 315
315 273 000 076 002 315 273 000 315 232 003 315 076 003 311 000
000 000 011 105 130 105 306 003 352 003 061 340 005 041 352 003
345 315 114 002 315 022 002 315 136 001 315 211 002 332 006 074
315 003 002 303 336 002 315 136 001 315 050 001 353 021 372 377
031 072 337 003 117 176 346 200 271 332 044 004 315 071 001 315
050 001 353 351 315 071 001 315 316 003 311 000 000 000 004 103
117 124 342 003 366 004 315 114 000 072 154 004 376 000 302 112
004 315 222 002 315 211 002 332 116 004 315 134 003 311 021 150
004 315 160 001 021 004 000 315 160 001 315 365 000 315 377 004
376 015 302 112 024 303 066 004 123 125 122 125 000 201 072 000
020 256 004 165 004 315 066 004 076 200 062 337 003 311 003 122
124 116 155 004 206 004 076 311 315 273 000 076 000 315 273 000
315 232 003 315 213 003 311 000 000 201 073 000 000 176 004 241
004 315 206 004 076 000 062 337 003 311 000 001 056 000 004 231
004 263 004 315 050 001 353 257 051 027 306 060 315 024 001 026
005 257 051 027 051 027 051 027 306 060 315 024 001 025 302 301
004 311 010 114 111 124 253 004 332 004 021 344 004 315 160 001
315 316 003 311 341 124 135 043 043 345 315 160 001 311 000 000
002 000 322 004 366 004 311 005 111 116 124 356 004 377 004 333
000 346 100 312 377 004 333 001 346 177 315 024 001 311 012 114
```

Figure 8. The object listing of the Initial Machine Code Dictionary.

CONVERS STANDARD HIGH LEVEL DICTIONARY

```
: JCOP 332 1, ;  : CONSTANT CODE LITERAL , RTN ;
   : VARIABLE CODE LITERAL, , RTN ;  CODE XCHG 353 1, RTN
    : ' UPDICT SEARCH @ ;  : JZOP 312 1, ;  : JMPOP 3Ø3 1, ;
    : JNZOP 3Ø2 1, ; CODE ( HERE ' INTTY COMPILE 376 1, 51 1,
    JNZOP , RTN  : RZ, 31Ø 1, ;
CODE DAD 353 1, 11 1, RTN  : + STK-BC STKDE DAD XCHG DE-STK ;
    ' COMPILE 21 + CONSTANT STATE CODE " HERE DUP ' INTTY COMPILE
 376 1, 12 1, RZ, 117 1, 72 1, STATE , 376 1, Ø 1, JZOP , ' BC-STK
 COMPILE ' 1, COMPILE JMPOP , RTN CODE 1- ' STKDE COMPILE 33 1,
    ' DE-STK COMPILE RTN  CODE COMPLEMENT ' STKDE COMPILE 172 1, 57 1,
 127 1, 173 1, 57 1, 137 1, ' DE-STK COMPILE RTN : MINUS COMPLEMENT 1+ ;


: ØPUSH Ø POP ;  : - MINUS + ;  CODE Ø< ' STKDE COMPILE 172 1,
    346 1, 2ØØ 1, 7 1, ' PUSH COMPILE ' ØPUSH COMPILE RTN
    : PRECEDENCE UPDICT SEARCH 6 - DUP @ 2ØØ + SWAP ! ;
   : STK-DICT LITERAL , ; PRECEDENCE STK-DHCT PRECEDENCE '
    : JNCOP 322 1, ; : IF ' ZERO? STK-DICT COMPILE JNCOP HERE Ø , ;
     PRECEDENCE IF  : ELSE JMPOP Ø , HERE SWAP ! HERE 1- 1- ;
    PRECEDENCE ELSE : THEN HERE SWAP ! ; PRECEDENCE THEN
   : = - IF Ø ELSE 1 THEN ;  : > - Ø< Ø = ;  : < SWAP > ;


: BELL 7 POP POP OUTTTY ;  PRECEDENCE "  : DO ' STKDE STK-DICT
 COMPILE 325 1, ' STKDE STK-DICT COMPILE 325 1, HERE ;  PRECEDENCE DO
 CODE TEST 3Ø1 1, 321 1, 341 1, 43 1, 345 1, 325 1, 3Ø5 1, 174 1,
    57 1, 147 1, 175 1, 57 1, 157 1, 43 1, 31 1, RTN
   : LOOP ' TEST STK-DICT COMPILE JCOP , 321 1, 321 1, ;  PRECEDENCE LOØ
 CODE +TEST 341 1, 3Ø1 1, 321 1, 33 1, 325 1, 3Ø5 1, 345 1,
    ' STK-BC COMPILE 353 1, 11 1, 3Ø1 1, 321 1, 361 1, 345 1, 325 1,
 3Ø5 1, RTN  : +LOOP ' +TEST STK-DICT COMPILE ' TEST STK-DICT COMPILE
 JCOP , 321 1, 321 1, ;  PRECEDENCE +LOOP   CODE DADSP 71 1, RTN
 : IC Ø STKDE XCHG DADSP XCHG DE-STK ;  : I IC 1Ø + @ ;
   : J IC 16 + @ ;  : K IC 24 + @ ;  CODE AND ' STKDE COMPILE
 ' STK-BC COMPILE 172 1, 24Ø 1, 127 1, 173 1, 241 1, 137 1,
    ' DE-STK COMPILE RTN  ' UPDICT 14 - CONSTANT DP
```

Figure 9a.

```
: ,CODE CODE HERE 1- 1- @ + HERE 1- 1- ! ;  : REMEMBER HERE 1- DUP
 CODE 41 1, , 42 1, DP , 52 1, 5 + , 42 1, DP 2 + , RTN ;
CODE 2* ' STKDE COMPILE 353 1, 51 1, 353 1, ' DE-STK COMPILE RTN
 CODE 2/ ' STKDE COMPILE 257 1, 172 1, 37 1, 127 1, 173 1, 37 1,
 137 1, ' DE-STK COMPILE RTN  CODE SWITCH ' STKDE COMPILE 353 1,
 125 1, 134 1, ' DE-STK COMPILE RTN  CODE SP 52 1, ' PUSH 14 +
     , 353 1, ' DE-STK COMPILE RTN  : UNDER 2* SP + @ SWITCH ;
 : MAX DUP 3 UNDER > IF SWAP THEN DROP ;
  : MIN DUP 3 UNDER < IF SWAP THEN DROP ;  CODE !! ' STKDE COMPILE
 ' STK-BC COMPILE 353 1,  161 1, RTN  : 1WORD DUP 3 UNDER @ SWAP
 !! 1+ SWAP 1+ SWAP ;  CODE WORDS ' STKDE COMPILE 325 1, HERE ' 1WORD
 COMPILE 321 1, 33 1, 325 1, ' DE-STK COMPILE ' ZERO? COMPILE
    JCOP , 321 1, ' DROP COMPILE ' DROP COMPILE RTN


CODE CVRT ' STK-BC COMPILE ' STKDE COMPILE 353 1, 76 1, 57 1, HERE
 74 1, 11 1, JCOP , ' OUTTTY COMPILE 353 1, ' DE-STK COMPILE RTN
 : .1Ø 2342Ø MINUS CVRT 2342Ø + 175Ø MINUS CVRT 175Ø + 144 MINUS
 CVRT 144 + 12 MINUS CVRT 12 + 6Ø + POP POP OUTTTY ;
   : 1ØCVRT Ø HERE @ 377 AND 1 DO DUP 2* 2* + 2* HERE I + @ 17 AND +
 LOOP ;  : DECIMAL 72 ' NUMBER STK-DICT 31 + !! ' 1ØCVRT STK-DICT
       ' NUMBER STK-DICT 43 + ! ' .1Ø STK-DICT ' . STK-DICT 2 - ! ;


: OCTAL 7Ø ' NUMBER STK-DICT 31 + !! ' OCTCVRT STK-DICT ' NUMBER
 STK-DICT 43 + ! ' ; STK-DICT 2Ø + DUP 2 + SWAP ! ;
   CODE DATAOUT 323 1, Ø 1, RTN : PORTOUT? ' DATAOUT STK-DICT 1+ !! ;
 : OUTDEVICE PORTOUT? POP POP DATAOUT ;
  CODE INDATA 333 1, Ø 1, RTN : PORTIN? ' INDATA STK-DICT 1+ !! ;
 : INDEVICE PORTIN? INDATA PUSH ; Ø VARIABLE SPECIAL
   : BEGIN-HERE HERE SPECIAL ! ;  : BEGIN 3Ø3 1, SPECIAL @ , ;
 : END 311 1, ; PRECEDENCE END PRECEDENCE BEGIN PRECEDENCE BEGIN-HERE
```

Figure 9b. The Standard High-Level Dictionary will be compiled

by the Initial Machine Code Dictionary.

CONFIGURING FOR ANOTHER MEMORY SIZE

1. Load the Initial Machine Code Dictionary as described in the conclusion of Article III. This can be loaded in any machine having at least 2K of memory. (The Initial Machine Code Dictionary resides in approximately 1.2K bytes.)
2. If less than 12K of memory is available, the value used to detect stack underflow and the initial stack pointer must be changed. If the user wishes to make use of more than 12K of continuous memory, these values must also be changed.
3. The stack underflow detection value (POP?) is located at address 333 octal. The value at this address should be changed to the 8 most significant bits of the memory bound plus two. For instance, the value of POP? for 12K of memory is $\phi61_8$. To configure for 8K, POP? should be changed to $\phi41_8$.
4. The stack pointer (SP) is located at address $3\phi7_8$ and $31\phi_8$. (Note that the SP is a 16 bit value with the least significant bits first.) This should be changed to the upper bound of continuous memory plus one. For instance, to change the present value of the stack pointer from 12K ($\phi\phi\phi_8$ and $\phi6\phi_8$) to 8K, the new stack pointer value should be $\phi\phi\phi_8$ at address $3\phi7_8$ and $\phi4\phi$ at address $31\phi_8$.

TABLE I

TERMINAL I/O AS PRESENTLY SUPPLIED
IN THE INITIAL MACHINE CODE DICTIONARY

'INTTY' Routine (starts at 2377_8)

```
HERE    IN STATUS     (333$_8$); Input current contents of flag
        STATUS PORT   (ϕϕϕ$_8$)  port (port ϕ)
        ANI 1ϕϕ       (346$_8$); Mask out all bits except bit 6
                      (1ϕϕ$_8$)  which is the data available flag
        JZ HERE       (312$_8$); Loop to 'HERE' if flag not set.
                      (377$_8$)
                      (ϕϕ4$_8$)
        IN DATA       (333$_8$); Input data from port one.
        DATA PORT     (ϕϕ1$_8$)
        ANI 177$_8$       (346$_8$); Strip parity bit.
                      (177$_8$)
        CALL OUTTTY   (315$_8$); Echo Character.
                      (ϕ24$_8$)
                      (ϕϕ1$_8$)
        RET           (311$_8$)
```

'OUTTTY' Routine (starts at 424_8)

```
HERE 1  PUSH PSW      (365$_8$); Push contents of 'A' reg. on stack.
        IN STATUS     (333$_8$); Input current contents of flag port
                      (ϕϕϕ$_8$)  (port ϕ)
        ANI 2ϕϕ       (346$_8$); Mask out all bits except bit 7 which
                      (2ϕϕ$_8$)  is the transmitter buffer port flag.
        JZ HERE       (312$_8$); Loop to 'HERE 1' until flag set.
                      (ϕ24$_8$)
                      (ϕϕ1$_8$)
```

POP PSW	(361_8); Get value back into 'A' register.
OUT DATA	(323_8); Output character to port one $(\phi\phi 1_8)$
RET	(311_8)

NOTE: Input routine, 'INTTY', can extend several bytes longer without any modification to the rest of the system.

TABLE II

number of routines are needed to successfully emulate the Initial Machine Code dictionary (IMCD) on other CPU's, it should be emphasized that each particular routine is logically rather simple. It is only when taken as a whole that the CONVERS system becomes so powerful.

An object listing of the Initial Machine Code dictionary (IMCD) is given in Figure 8. An octal loader should be used to load memory starting at octal 100. The initial Machine Code dictionary's starting address is 1752 octal. Once it is loaded and running, it will itself compile the rest of Standard CONVERS; a source listing is given as Figure 9. The version given here is for 12k of memory. To reconfigure for another memory size, the procedure outlined in Table I should be followed.

Table II lists the contents and memory location of the terminal I/O routines. These routines may also need to be reconfigured (or rewritten) depending on the user's particular terminal or interface.

This work was partially supported by the Office of Naval Research and by an Alfred P. Sloan Foundation Research Fellowship to M.B.D.

CONVERS DOS SYSTEM

For use in conjunction with the North Star Disk
System and CONVERS Initial Machine Code Dictionary
and the CONVERS Standard High Level Dictionary.

INTRODUCTION

The CONVERS Disk Operating System is designed for use with the North Star® disk system. It is possible to write a DOS system for any disk, in this case, the following routines must be rewritten:

DISK - INIT
MOTOR - START
TRACK - FIND
XREAD
XWRITE
TRACK - SECTOR

These routines will be described later, giving special discussion as to the requirements that each routine must meet if these routines are to be rewritten for another disk system.

The CONVERS-DOS (C-DOS) system has been written assuming a minimal disk system, i.e. a single drive. Due to the nature of C-DOS, other disk drives should not be needed except under the most taxing of situations.

GENERAL

The C-DOS system is designed as an interface between the floppy disk and the CONVERS general software system. Interaction with the disk is generally of two levels. The first level involves the transfer of specified memory contents to or from a block on the disk. A 'block' has been defined as 256_{10} bytes (neglecting the preamble, sync character and CRC character). Note that this block size is identical to the amount of data which one sector (at a particular track) of the North Star disk can contain. Thus to interact with the disk at this basic level, the memory location where data transfer is to take place must be the second number under the top of the stack, the block number must be the top number. The respective disk 'READ' or 'WRITE' routine is then executed. As an example, to write (out to the disk) the contents of memory starting at octal 1Ø,ØØØ to block 1ØØ octal, the following character string would be expected:

```
1ØØØØ 1ØØ WRITE
```

To 'read' from the disk, the analogous 'READ' routine is used:

```
11ØØØ 1Ø1 READ
```

The above character string, if expected, would deposit the contents of the disk block 1Ø1 to memory starting at 11ØØØ. Note that the direction of data transfer is referenced from the disk, i.e. data is read from or written onto the disk.

The 'WRITE' and 'READ' routines are used for binary data transfer to or from the disk. Using these routines, binary data or programs may be swapped back and forth. However, the most general use of the disk is for the storage of source programs. This level of disk access will be discussed in the next section.

SOURCE PROGRAM-STORAGE AND LOADING

Source programs are written to or loaded from the disk through two block buffers set aside specifically to handle disk source code transfers. The beginning location of these blocks is contained as the variable '/*BUFF'. To move the location of the buffers, this variable must be reinitialized to contain the new buffer address. To set the buffers at 6ØØØØ octal, for example, would require the following character string to be executed:

```
6ØØØØ /*BUFF !
```

Note that this is the standard way of reinitializing any variable. Two restrictions are placed on the location of the buffers; however, the buffers may not reside within the bounds of the CONVERS system and 512 continuous bytes must be allocated for buffer usage.

To place source code into the buffer, the 'PUT-BUFF' routine is used. The up arrow (↑) ends the filling of the buffer and returns control back to the CONVERS executive routine. As an example, suppose the user wishes to store the following source code into the buffer. The following would be entered on the terminal:

```
PUT-BUFF  :  SOUND  1Ø  1  DO  BELL  LOOP  ;  ;S  ↑
```

The definition of 'SOUND' is the required source program in this case. To display the contents of the buffer, the 'BUFF-DISP' routine is executed. To actually store the contents of the buffer onto the disk, the 'WR-VERIFY' routine is executed. The block number to which the data is to be transferred must be on top of the stack. For instance, to store the buffer contents at block 3ØØ, the following routine is executed:

```
3ØØ  WR-VERIFY
```

The 'WR-VERIFY' routine writes the buffer out to disk and reads it back to verify data transmission. If the user (or some program) wishes to neglect the verify step, the 'WR-BUFF' routine may instead be executed.

All source code blocks stored on disk must end with the ';S' definition being the last executable definition to appear in the block. Since the ';S' routine needs to be recognized, a space must follow the ';S' entry followed by the up arrow.

Source code stored on disk would appear exactly as if entered off the terminal, except, of course, the incorporation of the ;S and up arrow character strings. When using the 'PUT-BUFF' entry to initially enter source code to the buffer, 'PUT-BUFF' monitors the number of characters entered into the buffer. A non-fatal diagnostic message, '?7', will be output to the terminal when the buffer is about to be over-run. The user must end the buffer (with the ;S space and up arrow) in the next 11 characters entered.

LOADING SOURCE CODE BLOCKS FROM DISK

To load a source code block, the 'LOAD' entry is executed. The required block number must first be on the stack. To load block 3ØØ, for example, the following character string is executed:

```
3ØØ  LOAD
```

The source code is initially loaded into the block buffer whenever the 'LOAD' command is executed. The character input mode is changed from the terminal to the block buffer and the source code is executed or compiled as if it is coming from the terminal. The ';S' entry ends the execution or compilation of the source code. If the ;S entry is not encountered in the buffer, a fatal error is assumed and an error message '?8' is output to the terminal. Control is passed unconditionally to the executive after putting the system back into the terminal input mode.

LOADING MULTIPLE BLOCKS

Application dictionaries will reside in more than one block. For instance, the floating point package on our system in stored in about 45_{10} blocks on the disk. Obviously, it would be tedious to have to load each block individually. To eliminate this problem, a mechanism has been provided to allow one source block to load a number of other blocks. For instance, suppose source code blocks to control some experiment have been stored in blocks 1ØØ to 1Ø5 inclusive. The user might store the following source code in block 1Ø6 , for instance:

```
1ØØ  LOAD  1Ø1  LOAD 1Ø2  LOAD  1Ø3  LOAD  1Ø4  LOAD  1Ø5  LOAD  ;S  ↑
```

To load the whole application dictionary, it is now only necessary to load block 1Ø6. This block supervises the loading of the rest of the application dictionary.

The loading block (block 1Ø6 in the above example) must explicitly reference each block that is to be loaded. This means that a DO-LOOP may not be used to load a string of blocks. However, a DO-LOOP may be used to push on the stack a string of block numbers (in the reverse order from which they are to be loaded) and then a 'LOAD' command for each block number should follow. For example, block 1Ø6, in the above example, might contain the following:

```
:  EXPLOAD  5  Ø  DO  1ØØ  5  +  I  -  LOOP  ;  EXPLOAD  LOAD  LOAD  LOAD
                         LOAD  LOAD  LOAD  ;S  ↑
```

The above loading sequence assumes that block 1ØØ must be the first block to be loaded.

All users are strongly urged to read the descriptions of each routine supported by the C-DOS dictionary in the following section. Special notice should be made concerning the 'DISK-INIT' routine.

THE CONVERS-DOS (C-DOS) DICTIONARY

Each of the routines supported by the C-DOS dictionary will be described in turn. Routine names beginning with an asterisk (*) are routines that would have to be rewritten if another disk system other than the North Star® is to be used. Special notice should be taken of the comments concerning the 'DISK-INIT' routine if the user is adding the C-DOS system to the Standard CONVERS Dictionary.

*DISK-INIT

The 'DISK-INIT' routine initializes the disk to track zero and deposits the track number (zero) in address 1515 octal. This routine is called whenever the CONVERS system is booted up from the disk. To do this, the following simple routine has been added to the start of the Initial Machine Code Dictionary. Note that this routine may be destroyed once the disk has been initialized. (This may happen when vectored interrupt routines are employed since the 'jump to DISK-INIT' routine resides in vectored interrupt memory space.)

Memory Address	OPCODE OPERAND	COMMENTS
66_8	LXI SP	Initializes
67_8	IMCD ADDRESS	the stack pointer
70_8	OF STACK POINTER	
71_8	LXI HL	
72_8	ADDR OF EXECUTIVE	
73_8	ROUTINE	
74_8	PUSH H	Pushes address
75_8	JMP	of 'EXECUTIVE'
76_8	ADDR OF DISK-	on return stack
77_8	INIT ROUTINE	

*MOTOR-START

This routine starts the motors if they have not been started. If the motors are not on after start-up, a specified delay (one second) is executed to insure motors have come to speed. The read-write head is also engaged and a wait period (40 ms) is executed to ensure that the heads have engaged.

*TRACK-FIND

This routine calls the 'STK-BC' routine and steps to the track position found in the 'C' register. It also stores the track value in the track value storage location (1515_8).

*TRACK-SECTOR

This routine decodes the block number found on the stack such that the 'C' register holds the track number and the 'B' register holds the sector number. The decoded value is pushed back on the stack.

*XREAD

This routine 'reads' a sector off the disk and deposits the sector contents to the memory location found on the stack. The required sector value should be in the 'B' register. It is assumed that the heads are at the required track. 'XREAD' also pushes on the stack the 'status' value of the read operation. A zero indicates a successful 'read', one indicates a format error (sync character not found or drive not loaded) and two indicates a CRC error. The value that is pushed on the stack is an 8 bit value.

CHANGE-MODE

This routine changes the input mode back to the terminal input mode.

JMPEXEC

'JMPEXEC' unconditionally returns to the 'EXECUTIVE' after zeroing 'STATE'.

/*BUFF

This is a variable which holds the current buffer address.

TEMPCOUNT

'TEMPCOUNT' is used as a flag to determine if source code coming from disk should be loaded in the first (TEMPCOUNT = Ø) or the second (TEMPCOUNT > Ø) block buffer. See 'LOAD'.

READ

'READ' calls 'TRACK-SECTOR', 'MOTOR-START', 'TRACK-FIND' and 'XREAD' to initialize the reading operation. If a read error has occurred after 'XREAD' has completed, 'TEMPCOUNT' is zeroed and an error message is output. 'JMPEXEC' is called to unconditionally return to the 'EXECUTIVE', aborting the 'read' operation. The memory address and block number must be on the stack before 'READ' is executed.

BLOCK

'BLOCK' 'reads' the block whose number is on the stack into the first disk block buffer.

B-DISP

This routine outputs a string of ASCII characters from memory (indirectly addressed by the H & L registers) to the terminal. The routine ends upon detection of the up-arrow character (136_8).

BUFF-DISP

This routine displays the contents of the block buffer to the terminal.

P-BUFF

This routine deposits ASCII characters from the terminal to memory starting at the address in the H & L registers. It terminates upon detection of the up-arrow (↑) character. It outputs a diagnostic message when the user is about to over-run the buffer.

PUT-BUFF

'PUT-BUFF' calls 'P-BUFF' to allow the user to input ASCII characters to the block buffer.

SHOW

This routine displays the contents of all blocks inclusive between two block numbers on the stack. For instance, to display the blocks between 1ØØ and 1Ø5, the user would execute the following:

```
1Ø5  1ØØ  SHOW
```

WRITE

This routine calls 'TRACK-SECTOR', 'MOTOR-START', 'TRACK-FIND' and 'XWRITE' to initialize the disk write operation. The memory address and disk block number must first be on the stack. If the disk is write-protected, an error message (?6) is output to the terminal.

WRITE-VERIFY

'WRITE-VERIFY' is identical to 'WRITE', except the data is read back to verify data transmission.

DISP-BLOCK

This routine displays the contents of the block (whose number is on the stack) on the terminal.

COUNT

'COUNT' is a variable that holds the current buffer address during 'LOAD' operations.

1ØERROR

This routine, when executed outputs a '?8' to the terminal, zeroes

'TEMPCOUNT' and unconditionally jumps to the 'EXECUTIVE'. This routine is used in loading operations if a ';S' is not encountered when loading a block off disk.

BUFFIN

'BUFFIN' returns with the ASCII value addressed by 'COUNT' in the 'A' register. However, if the LSB's of the value stored at 'COUNT' equal 377_8, '1ØERROR' is called.

INBUFF

'INBUFF' pushes the contents of the H & L and D & E registers on the return stack and calls 'BUFFIN'. Upon the return of 'BUFFIN', the registers are restored.

MODE-CHG

This routine changes the input mode from the terminal to the block buffer input mode. It does this by changing the call address of 'UPDICT' from the 'INTTY' routine to the 'INBUFF' routine. (The routine 'CHANGE-MODE' returns this address back to 'INTTY'.)

LOAD

This routine 'loads' the block (block number must be on top of stack) and compiles and/or executes the source code found in the block.

;S

This routine ends loading of the current block.

WR-BUFF

This routine writes out to the disk from the block buffer. The block number must be on the stack.

WR-VERIFY

This routine is the same as 'WR-BUFF', but data transmission is verified by reading the block back in.

```
CODE DISK-INIT 16 1, 60 1, 315 1, 162 1, 351 1, 76 1, 0 1, 62 1,
 1515 , RTN  CODE MOTOR-START 72 1, 220 1, 353 1, 346 1, 20 1, 300 1,
  26 1, 62 1, 315 1, 320 1, 351 1, 72 1, 201 1, 353 1, 26 1, 15 1,
  315 1, 320 1, 351 1, RTN
CODE TRACK-FIND ' STK-BC COMPILE 171 1, 41 1, 1515 , 315 1, 144 1,
  351 1, RTN  CODE TRACK-SECTOR ' STKDE COMPILE 353 1, 21 1, 12 MINUS ,
  16 1, 377 1, HERE 14 1, 31 1, JCOP , 21 1, 12 , 31 1, 105 1,
 ' BC-STK COMPILE RTN
CODE xREAD ' STKDE COMPILE 353 1, HERE 315 1, 316 1, 351 1, 72 1, 60 1,
  353 1, 346 1, 17 1, 270 1, 302 1, , 21 1, HERE 12 + , 325 1, 21 1,
 400 , 325 1, 303 1, 114 1, 351 1, ' PUSH COMPILE RTN
: CHANGE-MODE ' INTTY STK-DICT ' UPDICT STK-DICT 21 + ! ;


CODE JMPEXEC 303 1, 237 , RTN
30000 VARIABLE /*BUFF 0 VARIABLE TEMPCOUNT
: READ TRACK-SECTOR MOTOR-START TRACK-FIND xREAD 0PUSH DUP 0 = IF
 DROP END THEN 0 TEMPCOUNT ! CHANGE-MODE 77 POP POP OUTTTY 63 + POP
  POP OUTTTY JMPEXEC ;
: BLOCK /*BUFF @ SWAP READ ;
CODE B-DISP HERE 176 1, ' OUTTTY COMPILE 376 1, 136 1, 310 1, 43 1,
  303 1, , RTN : BUFF-DISP /*BUFF @ STKDE xCHG B-DISP ;
 CODE 7ERR 76 1, 77 1, ' OUTTTY COMPILE 76 1, 67 1, ' OUTTTY COMPILE
 RTN CODE P-BUFF HERE ' INTTY COMPILE 376 1, 136 1, 167 1, 43 1, 310
  1, 76 1, 360 1, 275 1, 302 1, HERE 5 + , ' 7ERR COMPILE 303 1, , RTN
   : PUT-BUFF /*BUFF @ STKDE xCHG P-BUFF ;

: SHOW DO CRLF I DUP . CRLF BLOCK BUFF-DISP CRLF LOOP ;
CODE xWRITE ' STKDE COMPILE 353 1, 72 1, 20 1, 353 1, 346 1, 2 1,
 300 1, HERE 315 1, 316 1, 351 1, 72 1, 60 1, 353 1, 346 1, 17 1,
 270 1, 302 1, , 6 1, 0 1, 16 1, 17 1, 21 1, 0 1, 352 1, 72 1,
 4 1, 353 1, HERE 72 1, 20 1, 353 1, 346 1, 10 1, 312 1, , HERE
 32 1, 15 1, 302 1, , 36 1, 373 1, 32 1, HERE 176 1, 137 1, 32 1,
 176 1, 43 1, 250 1, 7 1, 107 1, 15 1, 302 1, , 130 1, 32 1, 257 1,
 RTN   : WRITE TRACK-SECTOR MOTOR-START TRACK-FIND xWRITE PUSH 0PUSH
  0 = IF END THEN  77 POP POP OUTTTY 66 POP POP OUTTTY ;
 : WRITE-VERIFY 2 UNDER 2 UNDER WRITE READ ;

: DISP-BLOCK BLOCK CRLF BUFF-DISP ;
0 VARIABLE COUNT  : 10ERROR 77 POP POP OUTTTY 70 POP POP OUTTTY
 CHANGE-MODE 0 TEMPCOUNT ! JMPEXEC ;
: BUFFIN COUNT @ DUP 377 AND 377 = IF DROP 10ERROR THEN DUP 1+
 COUNT ! 1@ POP ;
CODE INBUFF 345 1, 325 1, ' BUFFIN COMPILE 321 1, 341 1, RTN
 : MODE-CHG ' INBUFF STK-DICT ' UPDICT STK-DICT 21 + ! ;
 : LOAD TEMPCOUNT @ 0 = IF BLOCK 1 TEMPCOUNT ! /*BUFF @ COUNT !
 MODE-CHG ELSE COUNT @ TEMPCOUNT ! /*BUFF @ 400 + DUP COUNT !
 SWAP READ THEN ;
: ;S TEMPCOUNT @ 1 = IF CHANGE-MODE 0 TEMPCOUNT ! ELSE TEMPCOUNT
 @ COUNT ! 1 TEMPCOUNT ! THEN ;
: WR-BUFF /*BUFF @ SWAP WRITE ;  : WR-VERIFY /*BUFF @ SWAP 2 UNDER
 2 UNDER WRITE READ ;
```

CONVERS FLOATING POINT PACKAGE

For Use with CONVERS Standard High Level
Dictionary and Initial Machine Code Dictionary

Note !!!

1. The floating point source code shown below was taken off the disk in a block format. Note that the end of each 'block' is followed by the character string ';S ↑'. If the floating point source code is to be 'typed' into the computer (with CONVERS running) or is to be entered on mass storage, the above characters are ignored.

2. The source code shown references a routine called 'JMPEXEC' that is not contained in either the initial machine code dictionary or the standard high level dictionary. (It is, in fact, defined in our CONVERS disk system.) This was an oversight that was not caught when the floating point package was written or documented. Therefore, the user must first define this routine (as follows) before the floating point package is compiled:

```
CODE  JMPEXEC  3Ø3  1,  237  ,  RTN
```

Otherwise, the floating point package is totally compatible with standard 8Ø8Ø CONVERS.

INTRODUCTION

The CONVERS floating point package (C-FPP) is a group of software routines designed to implement many of the standard CONVERS operators in floating point form. It also allows floating point numbers on the stack to be stored as constants or variables in an analogous fashion to the treatment of integer numbers. The floating point package allows integer numbers to be converted to floating point numbers and vice versa. Thus, software may be written (defined) to control an interactive experiment such that integer values are used for I/O control, but data manipulation itself is handled using floating point numbers. The data resulting from this data manipulation is converted back to integer values and output to the corresponding output devices.

Documentation of the floating point package will be of two levels. The first level will be a functional description of the various routines the user would use to implement the floating point software. The second level involves a description of how the floating point routines work such that users of other CPU's can write versions for their particular machine.

GENERAL CONSIDERATIONS

The CONVERS floating point package (C-FPP) is a group of routines to implement floating point operations on six-digit floating point numbers. The six digits are stored as BCD numbers, two BCD numbers per byte. The most significant digit always resides in the six left-most four bits of each byte. A fourth byte is used to hold the exponent and the sign of the mantissa. The mantissa sign bit is the most significant bit with the bit 'set' indicating a negative mantissa. The exponent is stored in the seven remaining bits in an offset form. This means that an exponent of zero would be stored as 1ØØ octal, an exponent of minus one is stored as 77 octal, plus one as 1Ø1 octal and so on. The range of exponents is +63 decimal to -64 decimal.

Floating point numbers are input or output to the terminal under explicit command from the user, i.e. a floating point 'mode' cannot be chosen. To input the floating point mantissa the following routines may be used:

F	Input a positive mantissa
F+	Same as above
F-	Input a negative mantissa

As an example, to input the number -12.1246 the user would type the following:

```
F-    12.1246
```

Note that the number can be entered in any fashion, with the decimal point appearing anywhere in the number. (A decimal point does not have to appear.)

The exponent is entered by executing the following entries followed by the exponent value:

E	Input a positive exponent
E+	Same as above
E-	Input a negative exponent

To enter the 1.342 with an exponent value of negative 36, the following would be typed on the terminal by the user:

```
F 1.342  E-  36
```

Since the 'F' and 'E-' entries are definitions, they must be followed by a space. Likewise, a space is a delimiter to detect the end of the number character string, thus a space must also follow after the mantissa and the exponent value is entered. The floating point number once entered is pushed on the stack with the following BCD format:

```
                  __________      Least significant 2 digits
                  __________      Intermediate significance 2 digits
                  __________      Most significant 2 digits
Top of stack →    __________      Exponent and mantissa sign
```

To 'pop' the floating point number off the stack and have it displayed on the terminal, the 'F.' entry is executed. The number is output according to the following format:

```
±. Mantissa  E± Exponent
```

Note that the number is normalized, i.e. the mantissa will have a value between zero and one.

ARITHMETIC OPERATORS

Floating point arithmetic is executed between pairs of numbers on the stack using the following routines:

FADD	Adds two floating point numbers
FSUB	Subtracts two floating point numbers
FMUL	Multiplies two floating point numbers
FDIV	Divides two floating point numbers

Overflow or underflow, when detected, causes an error message (?9) to be output to the terminal. Control is passed unconditionally to the executive when such an error occurs. The two floating point numbers are left on the stack, however, such that the user can examine which pair of numbers caused the error.

VARIABLES AND CONSTANTS

Floating point variables and constants are handled in a way similar to integer manipulations. To define a constant for example, the 'FCONSTANT' routine is used. It executes exactly like the 'CONSTANT' entry, i.e. the floating value that will be named must be on the stack. After the 'FCONSTANT' entry, the new name of the constant must follow 'FZERO'. The routine is defined in C-FPP as follows:

```
F Ø FCONSTANT FZERO
```

Floating point variables are similarly defined in an analogous fashion to integer variables. The floating point number representing the initialized value must first be on the stack. The entry 'FVARIABLE' is entered followed by the user specified variable name. To define the floating point variable, 'FCOUNT', initialized to the value 1ØØ, the following would be executed:

```
F 1ØØ FVARIABLE FCOUNT
```

Floating point variables are reinitialized by the 'F!' entry. Floating point data stored as the floating point variable is accessed by the 'F@' entry. Both of these routines work much like the corresponding integer routines. For example, to reinitialize 'FCOUNT' to the value 1ØØØ, the following is executed:

```
F 1ØØØ FCOUNT F!
```

To push on the stack the current value of 'FCOUNT', the following is executed:

```
FCOUNT F@
```

STACK MANIPULATION ROUTINES

Several stack manipulation routines exist to handle floating point numbers. Each of these will be explained in turn.

FDROP - removes a floating point number from the stack.
FDUP - duplicates the top floating point number on the stack.
FSWAP - swaps the two top floating point numbers on the stack.
FUNDER - copies the specified floating point number under the stack back on top of the stack. For example, to copy the third floating point number under the stack, the following would be executed:

```
3 FUNDER
```

Note that the displacement (three in the above example) is an integer number.

FLOATING POINT COMPARISON ROUTINES

Three routines are available to test pairs of floating point numbers for certain conditions. These routines remove the pair of floating point numbers from the stack. These routines leave an integer value of one if the condition is true, a zero otherwise.

The routine 'F>' tests the second number under the stack to determine if it is greater than or equal to the top number. If the condition is true, a one is pushed on the stack, otherwise a zero.

The routine 'F < ' tests the second number under the stack to determine if it is less than or equal to the top number. Again, if the condition is true, a one is pushed on the stack, otherwise a zero.

The entry 'F=' tests the top pair of floating point numbers to determine if they are equal. If true, a one is pushed on the stack, otherwise a zero.

FLOATING - INTEGER CONVERSION

The entry 'FCONVERT', converts an integer number on the stack to a floating point value. The routine 'FLOAT' converts a floating point number back to an integer. If the floating point number represents a value greater than 65535_{10} (the largest value that an integer may represent), the octal value 177777 will be pushed on the stack regardless of the actual floating point value. Likewise, if the floating point number is less than zero, a zero will be pushed on the stack regardless of the actual value.

IMPLEMENTATION OF THE C-FPP

Several key concepts are behind the internal operation of the C-FPP. These concepts will be described in each respective section describing the operation of the C-FPP.

MANTISSA INPUT

The mantissa is input such that leading zeroes are ignored when the mantissa is greater than one. If the mantissa is less than one, leading zeroes must likewise be ignored; however, the exponent value must be decremented for each leading zero. A flow chart describing the logical floating point input operations is shown as Figure 1. A functional description of each mantissa input routine is described below.

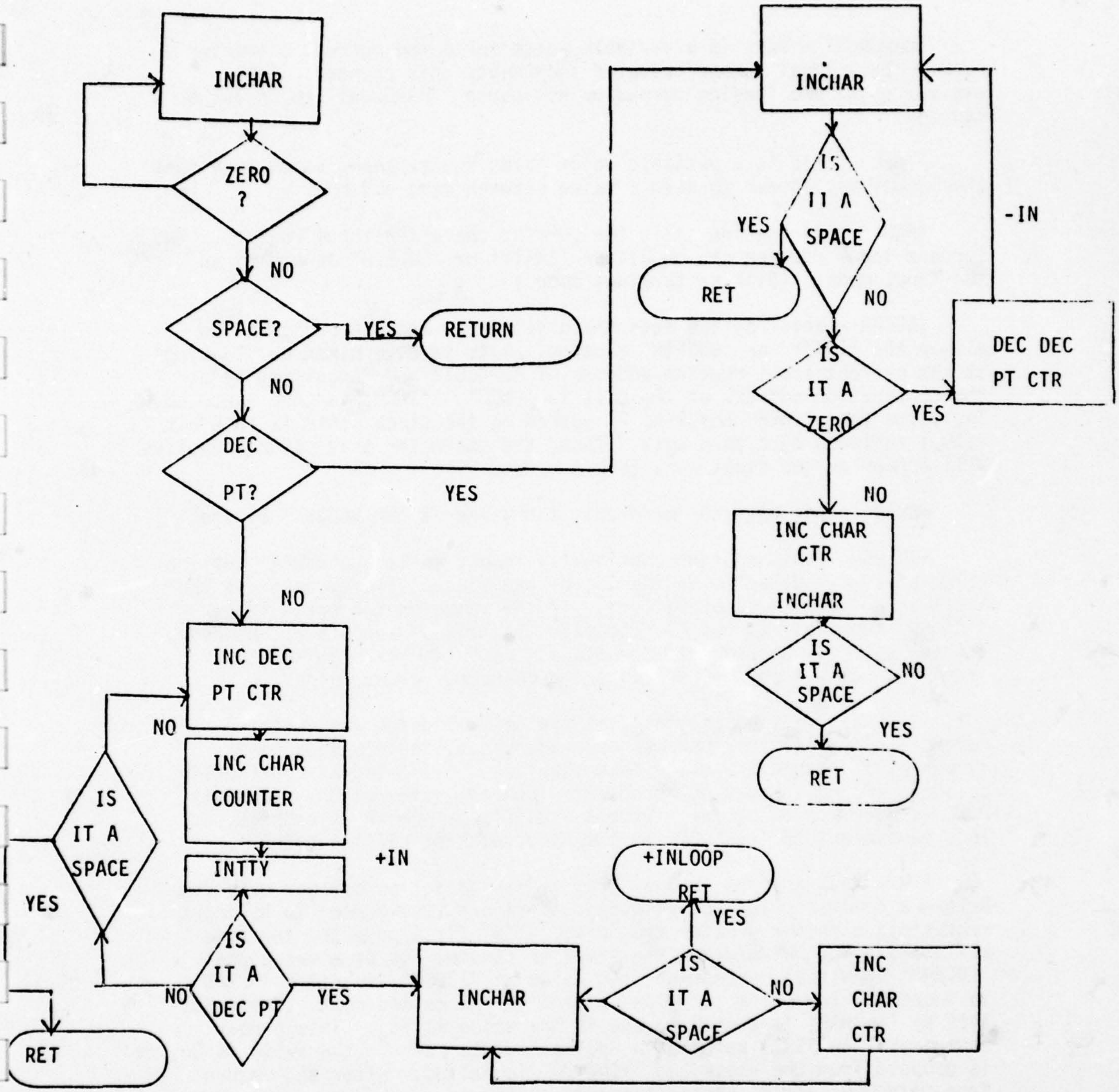


Figure 1. Flow chart to input a floating point mantissa.

CHARCOUNT - this is a variable which holds the current character count. Each legal number detected increments this counter. (The decimal point and leading zeroes do not cause 'CHARCOUNT' to be incremented.)

PTDEC - this is a variable which holds the exponent value such that the number may appear to have a value between zero and one.

INCH - this routine calls the current character input routine. The current input routine can be either 'INTTY' or 'BUFFIN' depending on the input mode. (Disk or terminal mode.)

INCHAR - this routine sets the destination address of 'INCH' to either the 'INTTY' or 'BUFFIN' routine. This is determined by 'looking' at the current input routine address of 'UPDICT' and depositing it as the destination address of the call in 'INCH'. 'INCH' is then executed. The value that 'INCH' supplies is pushed on the stack (this is an 8 bit value) followed by a zero byte. Thus, the character that 'INCH' supplies will appear on the stack as a 16 bit value.

+CHAR - this routine increments the value at 'CHARCOUNT' by one.

+INLOOP - this routine continually loops, calling 'INCHAR' until a value of 40$_8$ is detected on the top of the stack. (Forty octal is the ASCII value of a space character.) If the value on the stack (that 'INCHAR' supplied) is not 4Ø, the value is left on the stack. However, the top byte is 'popped' off the stack. Each time through the loop '+CHAR' is called which increments the character count value.

-IN -this routine assumes that the decimal point was detected before a significant number was entered. Thus, the floating point number will represent a value less than one. '-IN' loops, continually calling 'INCHAR'. Leading zeroes are ignored, however, the value at PDEC is decremented by one. Once a significant number is entered, a loop equivalent to '+INLOOP' is executed, without calling '+CHAR'.

+IN - this routine assumes that a significant number was detected before a decimal point was detected; therefore, the number to be input represents a number greater than one. '+IN' first pops the top byte off the stack (the value on the stack is assumed to be a value that 'INCHAR' supplied), increments the value at 'PTDEC' and calls '+CHAR' to increment the character count. 'INCHAR' is called next. The value left by 'INCHAR' is tested to see if the value is 56$_8$. This number represents the ASCII value of a decimal point. If the value is 56, it is dropped from the stack and '+INLOOP' is called. After the return from '+INLOOP', '+IN' unconditionally ends.

Assuming that the above value was not 56$_8$, it is tested for the 'space' character (40 octal). If true, this value is dropped and the routine ends. Otherwise, the routine loops back to input another character and performs the above testing functions over again.

FIN - this routine continually loops until a significant character input by 'INCHAR' is detected. Upon detection of a significant character, the '+IN' routine is called. If a decimal point is first encountered before a significant number is detected, the '-IN' routine is called. Upon detection of a space before a significant character or a decimal point is detected, the routine ends.

FPACKTEST - this routine tests the value at 'CHARCOUNT' for the following conditions. If 'CHARCOUNT' is equal to six, the routine ends. If the value at 'CHARCOUNT' is greater than six, the extra bytes are popped off the stack. If the value at 'CHARCOUNT' is less than six, zeroes are pushed on the stack such that the number of zeroes added to the stack and the 'CHARCOUNT' value equals six.

FALIGN - this routine AND's out all except the four least significant bits of two bytes on top of the stack. It switches the order of the two bytes and duplicates the number. This top number is right shifted by four bits. The two numbers are then added together. The new number generated by the above process is such that the BCD equivalent of the top two byte values on the stack are packed into one byte.

FPACK - this routine calls FPACKTEST to guarantee six bytes on the stack. It then packs the six bytes into three bytes in the required format. (MSD's on top of stack, etc.) The value of the exponent is then evaluated. If the value at 'PTDEC' is less than one, it is ANDed with 77 octal. Otherwise, 1ØØ octal is added to the number. Note that this has the effect of properly computing the offset exponent value. The MSB's of this number are then 'popped' off the stack. Thus, the four bytes left on the stack properly represent the floating point mantissa.

F - this routine zeroes the value of 'CHARCOUNT' and 'PTDEC', calls 'FIN' and 'FPACK' to allow the input of the mantissa and to properly 'pack' the number on the stack.

F+ - identical to the entry, 'F'.

F- - calls 'F' and sets the most significant bit of the exponent byte on the stack.

EXPONENT INPUT

The routines dealing with the input of the exponent value assume that the mantissa has already been input and properly packed on the stack.

E - this entry pushes a zero byte on the stack, calls 'UPDICT' and '1ØCVRT' to input the exponent and convert it. The converted value is added to the previously evaluated exponent supplied by the 'F' or 'F-' routine. The most significant byte is then popped off the stack.

E+ - same as 'E-.

E- - pushes a zero byte on the stack, duplicates the top number on the stack and 'AND's' the top number with 2ØØ octal. The two numbers on top of the stack are 'swapped'. This has the effect of 'saving' the mantissa sign value. 'UPDICT' and '1ØCVRT' are called. The exponent value that was input is subtracted from the previous exponent on top of the stack and this value is 'ANDED' with 177 octal. The new exponent value is added to the sign value number under the stack.

FLOATING POINT OUTPUT ROUTINES

Several routines are used to properly output a floating point number. Each of these routines will be described in turn.

.- - this routine outputs a minus sign character (-) on the terminal.

.+ - this routine outputs a plus sign (+) character on the terminal.

FSIGN - this routine evaluates the sign of the floating point number by pushing a zero byte on the stack, duplicating the number on the stack and ANDing it with 200 octal. If the resultant number is greater than zero (200), a negative sign is output by calling '.-'. Otherwise '.+' is called. The top byte on the stack is 'popped', restoring the original number.

.OUT - this routine outputs a decimal point on the terminal.

F1 - this routine duplicates the top number twice. The top number is 'ANDED' with 360 octal and right-shifted four bits. The ASCII offset, 60 octal, is added to this number and is popped and output to the terminal. This has the effect of outputting the most significant digit on the terminal.

The next number on the stack is ANDed with 17 octal and octal 60 is added to it. This number is popped off the stack and output to the terminal. This displays the next digit of the mantissa.

F2 - calls '2 UNDER' to get the least significant two bytes of the mantissa on top of the stack. The byte order is switched and 'F1' is called to display the next two digits. The remaining number left by 'F1' is then dropped off the stack.

F3 - '2 UNDER' and 'F1' is called to output the least significant two digits. The remaining number left by 'F1' is dropped. Note that the floating point number itself remains intact on the stack after 'F1', 'F2' and 'F3' execute.

EOUT - outputs an 'E' character on the terminal.

E. - this routine properly outputs the value of the exponent. A zero byte is added to the stack, ANDed with 177 octal, and duplicated. One-hundred octal is subtracted from the top number. If the resultant value is 1ØØ octal or greater, the exponent is assumed to be positive. The entries 'EOUT' and '.+' are called. The exponent is converted in an analogous fashion to the way the '.10' entry converts and outputs a number on the stack. However, only two digits need to be converted.

If the resultant number above is negative, a negative exponent is assumed. The '.-' entry and 'EOUT' entry are executed. 1ØØ octal is subtracted from the exponent value and 'MINUS' is executed. 1ØØ octal is subtracted from the exponent value and 'MINUS' is executed. The resultant exponent value is output in a manner similar to the '.10' entry as discussed above. Both branches discussed here remove the exponent byte.

F. - this entry calls 'FSIGN', '.OUT', 'F1', 'F2', 'F3', EOUT and 'E.' to properly output the floating point number. The remaining three bytes left on the stack are dropped from the stack.

FLOATING POINT ADD and SUBTRACT

Several routines are used to implement floating point addition and subtraction.

SPA - this constant holds the address of the stack pointer.

ALIGN - ADDR - this routine initializes the H & L and D & E register pairs such that the H & L registers point to the exponent address of the top floating number on the stack and the D & E registers point to the second number under the stack.

HCOPY - copies a byte string from the address pointed to by the H & L registers to the address pointed to by the D & E registers. The number of bytes to be copied is contained in the 'B' register.

FADDT - this routine adds the byte referenced by the H & L registers to the byte referenced by the D & E registers. The result is copied back to the location referenced by the D & E registers. The result of the addition must be 'decimal adjusted' after the addition. (The 8080 has a 'decimal adjust accumulator' instruction.) The H & L and D & E register pairs are decremented. If the 'B' register is not zero, the above series of operations are repeated.

INCSHIFT - this routine does a multibyte 'BCD' digit shift starting at the address in the H & L registers. Below is illustrated three bytes with the 'BCD' digits numbered:

```
5   6                (Memory increments
3   4                 upward)
1   2
```

After the shift, the byte contents would appear as follows:

```
4   5
2   3
D   1
```

The BCD digit 'D' above would be filled in with the contents of the 'D' register. The number of bytes to be shifted is held in the 'B' register upon entry into the 'INCSHIFT' routine.

FSUBT - this routine subtracts the byte referenced by the H & L registers from that referenced by the D & E registers. This byte subtraction must be a decimal subtraction. The way this is implemented is given in the flow chart in Figure 2. Note that the results of the auxiliary carry is used to generate the result. If another computer does not have this function, then a more complex method must be used to generate the auxiliary carry. (One way would be to perform the subtraction after masking out the left four bits. An auxiliary carry is then tested by examining the status of bit 4, assuming that bit Ø is defined as the least significant bit.)

FTEMP - this is used as a temporary memory block. Three zeroes are initially compiled into 'FTEMP'.

ALIGN - this routine calls 'ALIGN - ADDR'. It subtracts the byte referenced by the H & L registers from the byte referenced to by the D & E registers after masking out the sign bit. If the result is zero, the routine ends. If the result ended in a negative number, the result is complemented and incremented. The exponent value pointed to by the H & L registers is copied into the exponent value pointed to by the D & E registers. However, the original sign is saved. The D & E and H & L registers are also swapped.

The result of the exponent subtraction evaluated above (contained in the accumulator) is compared with the value, six. If the accumulator is greater than six, it is set equal to six. The following loop is executed. The number of loops made depends on the value of the accumulator. The H & L registers are first incremented to point to the most significant digit byte.

The loop discussed above consists of the following operations. The 'B' register is initialized with the value three, the 'D' register with zero. The contents of the H & L registers are saved by popping them on the return stack. The 'INCSHIFT' routine is called. Upon return, the H & L registers are restored. The accumulator value (stored in the 'e' register) is decremented and tested for zero. If not zero, the loop repeats again.

The result of the above series of operations is to align the floating point numbers such that the most significant digit of each floating point number has the required significance, depending on the value of the exponents. If the exponents differ by a value greater than five, it can be seen that one floating point number will be zeroed. The floating point number that is shifted depends on which number has the smallest exponent.

FNORM - this routine is used after the addition or subraction is performed to 'normalize' the second floating point number under the stack, i.e. to assure that the most significant digit of the result is non-zero. It continually does a four bit left shift until the most significant BCD digit contains a non-zero value. Before the shifting operation takes place, however, the floating point number is tested to see if any of the BCD digits are non-zero. If all are zero, the

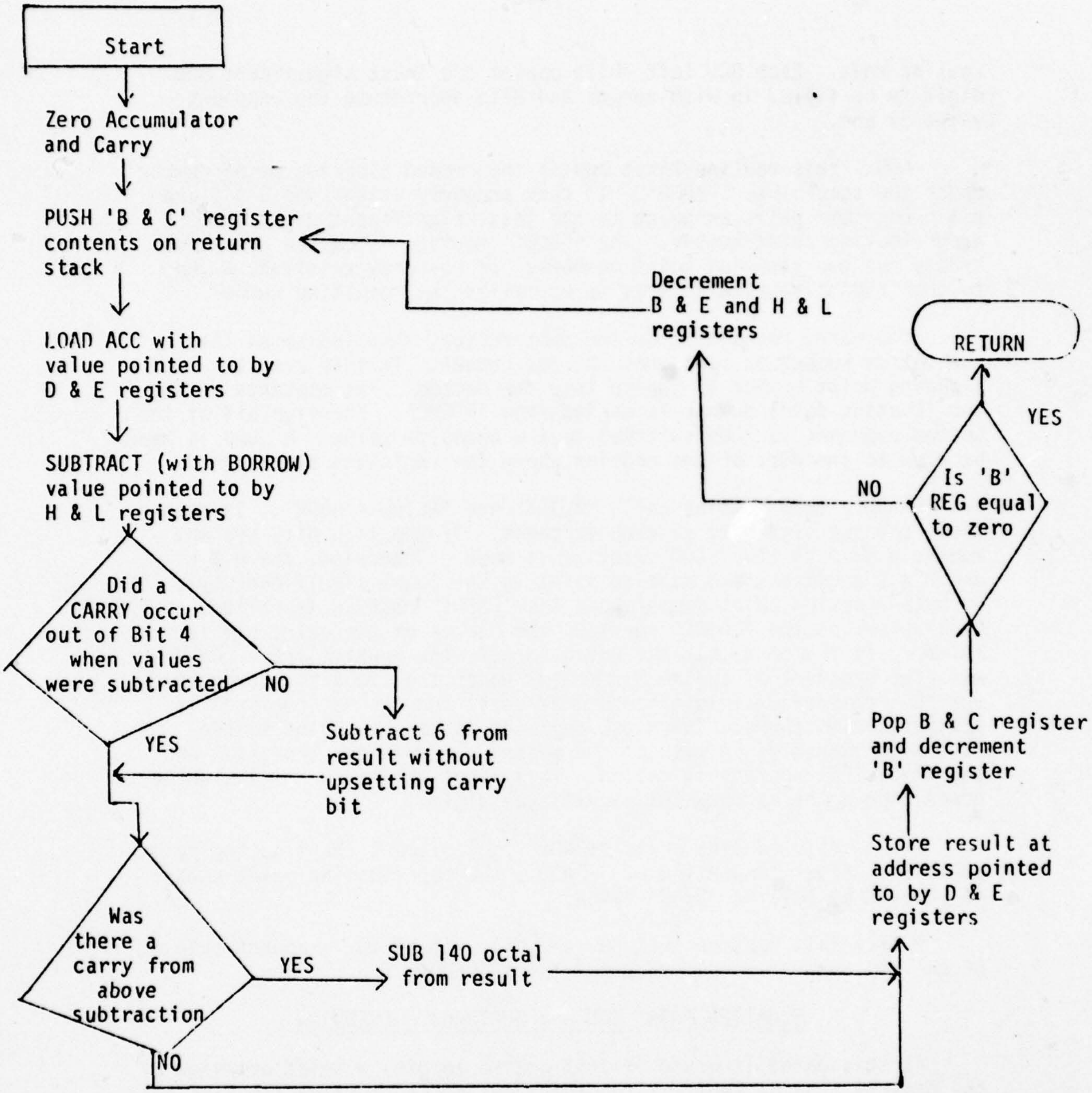

Start
Zero Accumulator and Carry
PUSH 'B & C' register contents on return stack
LOAD ACC with value pointed to by D & E registers
SUBTRACT (with BORROW) value pointed to by H & L registers
Did a CARRY occur out of Bit 4 when values were subtracted
NO
YES
Subtract 6 from result without upsetting carry bit
Was there a carry from above subtraction
YES
NO
SUB 140 octal from result
Store result at address pointed to by D & E registers
Pop B & C register and decrement 'B' register
Is 'B' REG equal to zero
YES
NO
RETURN
Decrement B & E and H & L registers

Figure 2. The 'decimal' subtract flow chart.

routine ends. Each BCD left shift causes the least significant BCD digit to be filled in with zeroes and also decrements the exponent value by one.

-ADD - this routine first copies the second floating point number under the stack into 'FTEMP'. It then properly aligns the D & E and H & L register pairs to point to the least significant digit byte of each floating point number. The 'FSUBT' routine is called which subtracts the two floating point numbers. If no-carry resulted, a jump to the 'FNORM' routine is made to normalize the resulting number.

If a carry resulted from the subtraction, this indicates that the bottom number is less than the top number. In this case the top floating point number is copied into the bottom. The contents of the top floating point number is copied from 'FTEMP'. The sign bit of the bottom exponent is then switched to the opposite value. A jump is made back up to the part of the routine where the registers are aligned.

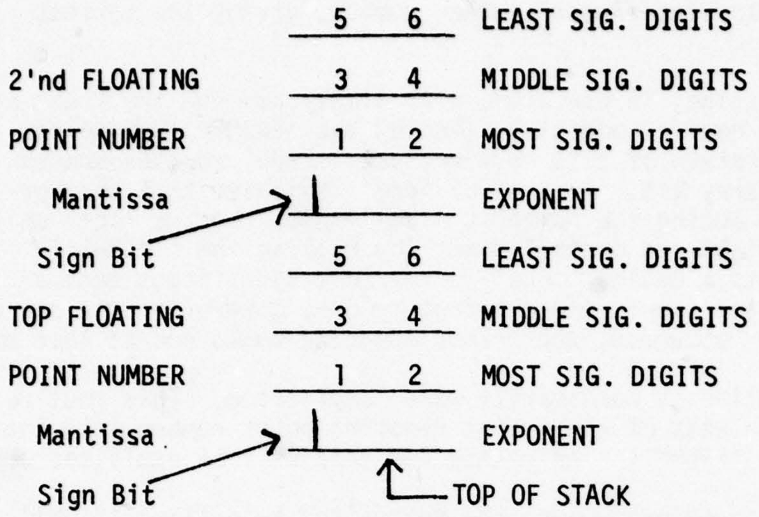
+FADD - this routine calls 'ALIGN' and "ALIGN - ADDR'. It then tests the two sign bits of each exponent. If the sign bits are not equal, a jump to the '-ADD' routine is made. Otherwise, the H & L and D & E registers are made to point to the least significant byte of both floating point numbers and the 'FADDT' routine is called. Upon return of the 'FADDT' routine, the status of the carry bit is tested. If the carry bit has not been set, the routine ends. Otherwise the exponent of the second number under the stack is incremented, the 'D' register is initialized to 20 octal and the 'B' register is initialized to three. The H & L register is made to point to the most significnat digit byte of the second number under the stack and the 'INCSHIFT' routine is called. This right-shifts the digits while depositing a one as the most significnat digit.

FADD - this routine calls '+FADD' to implement the floating point addition. After completion of '+FADD', the top floating point number is removed by calling 'DROP' twice.

FSUB - this routine switches the sign bit of the exponent byte of the top number on the stack and calls 'FADD'.

FLOATING POINT ADD and SUBTRACT - OVERVIEW

At this point it would be instructive to give a brief overview of the general flow of logic of the addition and subtraction floating point routines. First, a memory map of the system upon entry into this group of routines:



Note that the top of the stack is at the exponent byte of the top floating point number.

The 'FSUB' routine simply changes the value of the mantissa sign bit of the top floating point number. This has the effect of changing the 'sense' of the operation that the 'FADD' routine will execute. (The 'FSUB' routine calls the 'FADD' routine after changing the mantissa sign bit of the top number.) The 'FADD' routine determines which operation to perform by looking at both mantissa sign bits of the two floating point numbers and performs an 'EXCLUSIVE - OR' logical function on the two bit values. Remember that this function will result in a zero if the two values are equal, a one if they are unequal. A zero result would indicate that the numbers should be added, i.e., the sign of both mantissas are the same. A 'one' result indicates that the numbers should be subtracted. Thus, the 'FADD' routine either jumps to the '-ADD' routine or calls the 'FADDT' routine depending if the logical result of the 'EXCLUSIVE - OR' operation left a 'one' or a 'zero', respectively.

Before the 'FADDT' or '-ADD' routines can be executed, it is necessary to 'align' the floating point numbers. To do this, the 'FADD' (actually the '+FADD') routine calls the 'ALIGN' routine. This routine aligns one of the floating point numbers such that both numbers have the same significance relative to the exponent. For example, take the floating point numbers 1.Ø EØ and 1.Ø E1. If the mantissas of these numbers were added or subtracted, an obvious error would occur because the mantissa with the exponent of one represents a number ten times greater than the mantissa with an exponent of zero. To overcome this problem, the 'ALIGN' routine compares the magnitude of the exponents. If they are equal, nothing needs to be done and the routine ends. In the above example, the exponents differ by one; therefore, the mantissa with the smallest exponent is shifted 'right' one decimal place. If the exponent differs by greater than six, the mantissa with the smallest exponent is shifted right six decimal places which effectively zeroes the number. A zero will then be simply added or subtracted from the other number, giving the correct result.

The 'FADDT' routine, if executed, will simply add the two floating point numbers using decimal addition. (Review the 'FADDT' routine description.) On the return of this routine, the '+FADD' routine checks the status of the carry bit. A carry of 'one' indicates that an overflow occurred while adding the numbers. The '+FADD' routine right shifts the second number right one decimal place (by calling the 'INCSHIFT' routine) and deposits a decimal 'one' in the most significant decimal position. It also increments the exponent by one, generating the correct result. If no carry occurred, the '+FADD' routine would end at that point.

The '-ADD' routine is necessarily more complicated. This routine first copies the contents of the second floating point number into the temporary location, 'FTEMP'. The reason for this will be explained below.

The two floating point numbers are subtracted by calling 'FSUBT'. The status of the carry bit is now tested to see if a 'borrow' occurred. A borrow indicates that the top floating point number was greater than the second floating point number. To correct for this situation, the top floating point number is copied into the position held by the second floating point number, and the contents of the temporary location, 'FTEMP', is copied into the location of the top floating point number. The original exponent of the second floating point number is restored after changing the sense of the sign bit. Note that the above effectively switches the relative locations of the two floating point numbers. These numbers are subtracted as before.

The result of the subtraction must be normalized such that the most significant digit is non-zero. A branch to 'FNORM' is made. This routine returns if the result is totally zero. If a non-zero digit is detected, a series of left-decimal shifts are made such that the most significant digit is non-zero. Each left-digit shift also causes the exponent to be decremented.

FLOATING POINT MULTIPLY

Floating point multiplication is accomplished in the following manner. Consider the following problem:

$$\begin{array}{r} 35 \\ \text{X}\ \ 18 \\ \hline \end{array}$$

This problem assumes two digit floating point numbers; however, the arguments presented can be extended to any number of digits.

Multiplication can be accomplished by repetitive addition; this is exactly what this floating point package does. The top number (35) is added to itself the number of times indicated by the first digit of the multiplier (8):

```
  35
   8
 ---
 280
```

Let's now shift the multiplicand left one digit and 'multiply' the new multiplicand by the next multiplier digit:

```
 350
   1
 ---
 350
```

Now add the two results:

```
 280
 350
 ---
 630
```

The sum is 630, which is the required answer. Thus, the problem of multiplication is handled by successively adding the multiplicand to itself the number of times of each multiplier digit. The multiplicand is left shifted, and the new multiplicand is added to itself the required number of times according to the value of the next significant digit of the multiplier. The sums are added and the process repeated, depending on the number of digits. This process of generating the answer to a multiplication problem is the way the CONVERS 'FMUL' routine works.

FLOATING POINT MULTIPLY

Each of the routines supporting the floating point multiply function will be described here.

DCRSHIFT - this routine right shifts decimal digits, starting at the address pointed to by the H & L registers. The contents of the 'D' register will be copied into the most significant digit position. The number of memory bytes that are to be shifted should be in the 'B' register upon entry into this routine. Below is a memory map, before and after the shifting operation, assuming a four byte shift:

1	2	Memory	D	1
3	4	Extends	2	3
5	6	Upward	4	5
X	Y		6	X
BEFORE			AFTER	

F - this routine is the 'heart' of the floating point multiply function. It first zeroes out the first $9_{1Ø}$ locations in FTEMP and copies the contents of the mantissa of the second floating point number into the next three locations in FTEMP, most significant digits first. The 'C' register is initialized to the value six. At this point, the D & E registers are pointing to the 'high end' of the temporary buffer 'FTEMP', (FTEMP + 12_{10})

and the H & L registers are pointing to the least significant digit byte of the top floating point number. A 'master loop' begins execution at this point.

The 'C' register is copied into the 'A' register which is right shifted. This will either set or reset the carry bit, depending on the previous value (before shifting) of the least significant bit of the 'A' register. The accumulator is then loaded with the value pointed to by the H & L registers. This is the least significant digit byte of the top floating point number at this point. If the carry bit is not set, the left four bits of the accumulator are zeroed. Otherwise the right four bits are zeroed and the accumulator is right shifted four places. The accumulator is now stored in the 'B' register. Note that the 'B' register will hold at this point the number of additions that the multiplicand should be added with itself. Both the contents of the B & C and H & L registers are pushed on the return stack. A copy of the 'B' register is also made into the 'C' register. The 'B' register is checked for zero. If it is, the addition loop is skipped below.

The addition loop consists of initializing the D & E registers to 'FTEMP' + 5, and the H & L registers to 'FTEMP' + 11_{10} . The 'B' register is loaded with the value six and 'FADDT' is called. The 'C' register is decremented; if it is not zero, the addition loop is repeated again. Otherwise it ends.

The 'D' register is loaded with zero, the 'B' register with the value six, and the H & L registers are loaded with the value of 'FTEMP' + 11_{10}. The routine 'DCRSHIFT' is called. This has the effect of left-digit shifting the multiplicand which was described in the beginning of this section.

The B & C and H & L registers are restored by popping them off the return stack. The 'C' register is decremented and if zero, the routine ends. Otherwise the 'C' register is copied into the 'A' register which is right shifted. If the carry bit is set, a jump is made again back to the start of the master loop.

Note in the above description that the contents of the 'C' register determine whether the left or right digit of each byte is to be the multiplier. If the 'C' register is even, the right digit is multiplied, otherwise the left. Each time through the master loop, the 'C' register is decremented. This also has the effect of decrementing the H & L registers only when the 'C' register is even. Thus the master loop must be executed twice before the next higher pair of multiplier digits are operated upon.

CKFACC - this routine checks the second floating point number for a zero condition. If this number is zero, it drops the top floating point number from the stack and returns with the zero flag set.

GETEXP - this routine returns with the value of the second floating point exponent in the 'C' register and the value of the top number in the 'B' register. In both cases, the sign bit of the mantissa is zeroed. This routine also determines the sign of the result by 'EXCLUSIVE-ORING' the two sign bits. The result of the EXCLUSIVE-OR operation is stored back into the sign bit of the second floating point number after zeroing all other bits.

FERROR - this routine will output a (?9) on the terminal and calls 'JMPEXEC'.

F*NORM - this routine checks to see if the most significant digit in the temporary buffer 'FTEMP' is non-zero. (This digit is at the first address of 'FTEMP'.) If non-zero, the routine ends. Otherwise, the 'D' register is zeroed, the H & L registers are set to point to 'FTEMP' + 3, and 'DCR SHIFT' is called. The exponent of the second floating point number is also decremented. This has the effect of normalizing the result of the multiplication.

FTARGET - this routine transfers the result of the multiplication in the temporary buffer, 'FTEMP', into the second floating point number. The entry, 'DROP', is called twice to remove the top number.

FMUL - this routine first calls 'CKFACC' and ends if the zero bit was set. Otherwise, 'GETEXP' is called and the exponents are then added together in the accumulator. To re-normalize the exponents, 300 octal is added to the result and it is saved in the 'C' register.

To determine if an overflow or underflow condition resulted, the accumulator is left shifted. If a carry results, 'FERROR' is called. The sign bit of the second floating point number is added to the value in the 'C' register and the result is copied back. This is the new computed exponent. The routines '*F*', 'F*NORM' and 'FTARGET' are called. At the conclusion of these routines, the D & E registers point to the memory byte after the least significant digit byte of the second floating point number.

The routine must now round off the answer. It does this by loading the value at 'FTEMP' + 3 into the accumulator and adds 120 octal to it. It then decimal adjusts the accumulator. If no carry resulted, the digit was less than 5 and the routine ends. Otherwise, each byte (starting with the least significant digit byte) has zero added to it. The carry bit is also added. (The 8080 has an addition with carry instruction.) The byte is then decimal adjusted and the procedure is repeated for each of the other two bytes.

FLOATING POINT DIVISION

Floating point division is different from multiplication in that division can be thought of as repetitive subtraction instead of addition. As an example:

$$212\,\overline{)689}$$

To perform the division, we could count the number of times that 212 can be subtracted from 689 while leaving a positive difference. In the above example, three subtractions can be performed:

```
   689
 - 212
   477
 - 212
   265
 - 212
 +  53
```

Therefore, the first digit of the dividend is three. Now lets shift the positive difference left one digit and repeat the process to get the next digit of the dividend:

```
 + 530
 - 212
   318
 - 212
 + 106
```

This time 212 can be subtracted twice, therefore, the next digit is two. Let's repeat for a third time:

```
 +1Ø60
 - 212
   848
 - 212
   636
 - 212
   424
 - 212
   212
 - 212
   000
```

This time 212 can be subtracted five times. Therefore, the dividend is 3.25. Since both numbers are normalized before the division starts, the position of the decimal point can be assumed.

In cases where the divisor is larger than the quotient:

```
314 ⟌162
```

The divisor cannot be subtracted into the quotient without giving a negative result. Therefore the first digit of the dividend is zero and we repeat the process as before. After two more digits of the dividend have been found, the question is asked, is the first digit a zero? If yes, the exponent of the result is decremented and the process repeated for one more digit. Thus, using this approach, three digits of significance are guaranteed.

To 'round off' the dividend, the process could be repeated one more time. If the extra digit is greater than four, one is added to the quotient.

The above logic sequences are what takes place in the CONVERS floating point division routine. The divisor is continually subtracted from the quotient until a negative number is generated (the carry flag, indicating a borrow, has been set). Since a negative number has been generated, the routine must add the divisor to the difference to generate a positive number. The number of subtractions is stored in the 'B' register.

/FSTORE - this routine is used to copy the contents of the digit bytes of the two floating point numbers to the temporary buffer 'FTEMP' in the following format:

BYTE CONTENT	MEMORY ADDRESS
ZEROED	FTEMP
MSD of 2nd NUMBER	+ 1
NEXT MSD of 2nd NUMBER	+ 2
LSD of 2nd NUMBER	+ 3
ZEROED	+ 4
MSD of TOP NUMBER	+ 5
NEXT MSD of TOP NUMBER	+ 6
LSD of TOP NUMBER	+ 7
	+ 8
	+ 9
ZEROED (this is a flag byte)	+10

FROUN - this routine is used to round off the second floating point number after division has taken place. It assumes that the 'B' register holds the '7th digit' which is tested to see if it is greater than or equal to five. If so, the carry bit is set and 'one' is added to the second floating point number in a process similar to the way the 'FMUL' routine rounds off the second floating point number after multiplication.

/F/ - this entry is the 'heart' of the floating point division function. It consists basically of three loops as described below. The first operation is to initialize the 'C' register with the value of six.

The 'outer loop' begins by initializing the 'B' register with the value of 377 octal. Now begins the 'inner loop'. The D & E registers are set to point to 'FTEMP' + 7. The B & C registers are pushed on the return stack. The 'B' register is initialized to four and the 'FSUBT' routine is called. The B & C registers are popped off the return stack and the 'B' register is incremented. If no carry resulted from the subtraction, a jump is made back to the start of the inner loop. At the end of this loop, the 'B' register holds the digit count of the first 'round' of the division.

The value of the flag ('FTEMP' + $1Ø_{10}$) is tested, if it is not zero, the routine ends. Otherwise the B & C registers are stored on the return stack and the 'C' register is initialized to three. The H & L registers are made to point to the least significant digit byte of the second number on the stack. The 'most inner' loop begins below.

The value pointed to by the H & L registers is loaded into the accumulator. A copy is made into the 'D' register. The left four bits are masked and the accumulator is rotated left four bits. The contents of the 'B' register is added to the accumulator. The accumulator is copied back into the byte pointed to by the H & L registers. The accumulator is loaded with the contents of the 'D' register and the right four bits are masked. The accumulator is rotated right four bits. The result is saved in the 'B' register. The H & L registers are decremented. The 'C' register is decremented and if it is not zero a jump is made to the start of the 'most inner' loop.

Otherwise, the B & C registers are restored by popping them off the stack. The D & E registers are made to point the 'FTEMP' + 3 and the H & L registers to 'FTEMP' + 7. The 'B' register is initialized to four and the 'FADDT' routine is called.

The 'B' register is again initialized to four and the H & L registers are made to point to 'FTEMP' + 3. The 'D' register is zeroed and the 'DCRSHIFT' routine is called. The 'C' register is decremented and if not zero, a jump is made to the start of the outer loop. Otherwise the D & E registers are made to point to the most significant digit byte of the second floating point number. This byte is 'AND'ed against 360 octal to determine if the most significant digit is zero. If so, the exponent is decremented, the 'C' register is initialized to one, and a jump is made to the start of the outer loop.

If the most significant digit is not zero, the flag at 'FTEMP' + 10_{10} is set to one and a jump is made to the start of the outer loop.

If the most significant digit is not zero, the flag at 'FTEMP' + 10_{10} is set to one and a jump is made to the start of the outer loop. This last time through the loop allows the 'B' register to contain the 'seventh digit' so that the quotient can be properly rounded off.

FDIV - this routine calls 'CKFACC' to determine if the second floating point number is zero. If so, the routine ends. Otherwise, the 'GETEXP' routine is called. On return of this routine the top floating point number is tested to see if it is zero. If it is, the 'FERROR' routine is called. If not zero the 'B' register containing the top number exponent is subtracted from the 'C' register containing the second number exponent. To normalize the exponent, 100 octal is added and the result is incremented. The result is stored in the 'C' register. To test for overflow or underflow, the result is left shifted. If a carry resulted, the 'FERROR' routine is called. Otherwise, the contents of the 'C' register is added to the sign bit of the exponent byte of the second floating point number. The '/FSTORE', '/F/', and 'FROUN' routines are called. The top floating point number is removed by calling 'DROP' twice.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
Ø VARIABLE CHARCOUNT Ø VARIABLE PTDEC CODE INCH 315 1, Ø ,
RTN : INCHAR ' UPDICT STK-DICT 21 + @ ' INCH STK-DICT 1+ ! INCH PUSH
 ØPUSH ; : +CHAR CHARCOUNT @ 1+ CHARCOUNT ! ; : +INLOOP BEGIN-GERE
 INCHAR DUP 4Ø = IF DROP END THEN +CHAR POP BEGIN ; ;S ↑

: -IN DROP BEGIN-HERE INCHAR DUP 4Ø = IF DROP END THEN DUP
 6Ø = IF DROP PTDEC @ 1- PTDEC ! BEGIN THEN POP BEGIN-HERE +CHAR INCHAR
  DUP 4Ø = IF DROP END THEN POP BEGIN ; ;S ↑

: +IN POP BEGIN-HERE PTDEC @ 1+ PTDEC ! +CHAR INCHAR DUP
 56 = IF DROP +INLOOP END THEN DUP 4Ø = IF DROP END THEN POP BEGIN ;
  : FIN BEGIN-HERE INCHAR DUP 6Ø = IF DROP BEGIN THEN DUP 4Ø = IF DROP
  END THEN DUP 56 = IF -IN ELSE +IN THEN ; ;S ↑

: FALIGN 7417 AND SWITCH DUP 2/ 2/ 2/ 2/ + ;
: FPACKTEST CHARCOUNT @ 6 > IF CHARCOUNT @ 6 = IF END ELSE CHARCOUNT
  @ 6 - 1 DO POP LOOP END THEN ELSE 6 CHARCOUNT @ - 1 DO ØPUSH LOOP
 THEN ;  ;S ↑

: FPACK FPACKTEST FALIGN SWAP FALIGN POP SWITCH POP SWAP
 FALIGN POP PTDEC @ DUP Ø< IF 77 AND ELSE 1ØØ + THEN POP ;
 : F Ø Ø CHARCOUNT ! PTDEC ! FIN FPACK ; : F+ F ;
: F- F ØPUSH 2ØØ + POP ;  ;S ↑

: .- 55 POP POP OUTTIY ;  : .+ 53 POP POP OUTTTY ;
: FSIGN ØPUSH DUP 2ØØ AND IF .- ELSE .+ THEN POP ;
: .OUT 56 POP POP OUTTTY ; : F1 DUP DUP 36Ø AND 2/ 2/ 2/ 2/ 6Ø +
 POP POP OUITTY 17 AND 6Ø + POP POP OUTTTY ; ;S ↑

: F2 2 UNDER SWITCH F1 DROP ;  : F3 2 UNDER F1 DROP ;
: EOUT 1Ø5 POP POP OUTTTY ; : E. ØPUSH 177 AND DUP 1ØØ > IF EOUT 1ØØ
  - .+ 12 MINUS CVRT 12 + 6Ø + POP POP OUTTTY ELSE EOUT .- 1ØØ - MINUS
  12 MINUS CVRT 12 + 6Ø + POP POP OUTTTY THEN ; ;S ↑

: F. FSIGN .OUT F1 F2 F3 E. DROP POP ; : E ØPUSH UPDICT
1ØCVRT + POP ; : E+ E ; : E- ØPUSH DUP 2ØØ AND SWAP UPDICT 1ØCVRT
 - 177 AND + POP ;  ;S ↑

' PUSH 14 + CONSTANT SPA CODE ALIGN-ADDR 52 1,
SPA , 124 1, 135 1, 23 1, 23 1, 23 1, 23 1, RTN CODE HCOPY HERE 176 1,
  22 1, 43 1, 23 1, 5 1, 3Ø2 1, , RTN CODE FADDT 257 1, HERE 32 1,
 216 1, 47 1, 22 1, 53 1, 33 1, 5 1, 3Ø2 1, , RTN ;S ↑
```

```
CODE INCSHIFT HERE 176 1, 137 1, 346 1, 360 1, 37 1, 37 1,
 37 1, 37 1, 262 1, 167 1, 173 1, 346 1, 17 1, 27 1, 27 1, 27 1, 27 1,
  127 1, 43 1, 5 1, 302 1, , RTN :S ↑

CODE [illegible] 157 1, HERE 305 1, 32 1, 236 1, 365 1, 301 1, 171
 1, 346 1, [illegible] 1, 302 1, HERE 6 + , 170 1, 326 1, 6 1, 107 1, 171 1,
  37 1, 302 1, HERE 6 + , 170 1, 326 1, 143 1, 107 1, 305 1, 361 1,
  [illegible] 1, 301 1, 53 1, 33 1, 5 1, 302 1, , RTN :S ↑

: FTEMP 0 3 0 ; CODE ALIGN ' ALIGN-ADDR COMPILE 32 1, 346 1,
 177 1, 107 1, 176 1, 345 1, 177 1, 117 1, 170 1, 221 1, 310 1, 362 1,
  HERE 14 + , 57 1, 74 1, 107 1, 32 1, 346 1, 200 1, 261 1, 22 1, 170 1,
  353 1, ;S ↑

376 1, 6 1, 332 1, HERE 4 + , 76 1, 6 1, 117 1, 43 1, HERE
 6 1, 3 1, 26 1, 0 1, 345 1, ' INCSHIFT COMPILE 341 1, 15 1, 302 1, ,
  RTN ;S ↑

CODE FNORM HERE ' ALIGN-ADDR COMPILE 353 1, 43 1, 176 1, 345 1,
 360 1, 340 1, 176 1, 43 1, 266 1, 43 1, 266 1, 310 1, 6 1, 3 1, 26 1,
  0 1, HERE 176 1, 117 1, 346 1, 17 1, 27 1, 27 1, 27 1, 27 1, 262 1,
  167 1, 171 1, ;S ↑

346 1, 360 1, 37 1, 37 1, 37 1, 37 1, 127 1, 53 1, 5 1, 302 1,
 , 55 1, 303 1, , RTN CODE -ADD ' ALIGN-ADDR COMPILE 353 1, 21 1,
  ' FTEMP , 6 1, 4 1, ' HCOPY COMPILE HERE ' ALIGN-ADDR COMPILE 43 1,
  43 1, 43 1, 23 1, 23 1, 23 1, ;S ↑

6 1, 3 1, ' FSUBT COMPILE 322 1, ' FNORM , 345 1, 325 1,
32 1, 117 1, 6 1, 4 1, ' HCOPY COMPILE 341 1, 321 1, 171 1, 356 1,
  200 1, 167 1, 41 1, ' FTEMP , 6 1, 4 1, ' HCOPY COMPILE 303 1, , RTN
  ;S ↑

CODE +FADD ' ALIGN COMPILE ' ALIGN-ADDR COMPILE 32 1, 256 1,
 372 1, ' -ADD , 43 1, 43 1, 43 1, 23 1, 23 1, 23 1, 6 1, 3 1,
 ' FADDT COMPILE 32 1, 353 1, 64 1, 26 1, 20 1, 6 1, 3 1, 43 1,
 ' INCSHIFT COMPILE RTN ;S ↑

: FADD +FADD DROP DROP ;
CODE FSUB ' ALIGN-ADDR COMPILE 176 1, 356 1, 200 1, 167 1, ' FADD
 COMPILE RTN ;S ↑

CODE DCRSHIFT HERE 176 1, 346 1, 360 1, 37 1, 37 1, 37 1, 37
 1, 137 1, 176 1, 346 1, 17 1, 27 1, 27 1, 27 1, 27 1, 262 1, 167 1,
  123 1, 53 1, 5 1, 302 1, , RTN :S ↑

CODE *F* 43 1, 43 1, 43 1, 345 1, 41 1, ' FTEMP , 6 1, 11 1,
 HERE 66 1, 0 1, 43 1, 5 1, 302 1, , 23 1, 353 1, 6 1, 3 1, ' HCOPY
  COMPILE 16 1, 6 1, 341 1, HERE 171 1, 37 1, 176 1, 332 1, HERE 7 +
  , 346 1, 17 1, 303 1, HERE 10 + , :S ↑

346 1, 360 1, 17 1, 17 1, 17 1, 17 1, 107 1, 345 1, 305 1,
 110 1, 267 1, 312 1, HERE 21 + , HERE 21 1, ' FTEMP 5 + , 41 1,
 ' FTEMP 13 + , 6 1, 6 1, ' FADDT COMPILE 15 1, 302 1, , 26 1, 0 1,
  6 1, 6 1, 41 1, ' FTEMP 13 + , :S ↑

' DCRSHIFT COMPILE 301 1, 341 1, 15 1, 310 1, 171 1, 37 1,
 332 1, DUP , 53 1, 303 1, , RTN CODE CKFACC ' ALIGN-ADDR COMPILE
  23 1, 32 1, 346 1, 377 1, 365 1, 302 1, HERE 10 + , ' DROP DUP
 COMPILE COMPILE 361 1, 33 1, RTN :S ↑

CODE [illegible] 32 1, 117 1, 256 1, 346 1, 200 1, 22 1, 171 1,
 346 1, 177 1, 117 1, 176 1, 346 1, 177 1, 107 1, RTN
: FERROR 77 POP POP OUTTTY 71 POP POP OUTTTY JMPEXEC ;
;S ↑
```

```
CODE F*NORM 41 1, ' FTEMP , 176 1, 346 1, 36Ø 1, 3ØØ 1, 6 1,
 4 1, 43 1, 43 1, 43 1, 26 1, Ø 1, ' DCRSHIFT COMPILE ' ALIGN-ADDR
  COMPILE 32 1, 75 1, 22 1, RTN CODE FTARGET ' ALIGN-ADDR COMPOIE 23 1,
  41 1, ' FTEMP , 6 1, 3 1, ' HCOPY COMPILE :S ↑

' DROP DUP COMPILE COMPILE RTN CODE FMUL ' CKFACC COMPILE
 317 1, ' GETEXP COMPILE 17Ø 1, 241 1, 3Ø6 1, 3Ø2 1, 117 1, 27 1,
  322 1, HERE 5 + , ' FERROR COMPILE 32 1, 261 1, 22 1, ' *F* COMPILE
  ' F*NORM COMPILE ' FTARGET COMPILE 176 1, :S ↑

72 1, ' FTEMP 3 + , 3Ø6 1, 1ØØ 1, 47 1, 3ØØ 1, 33 1, 353 1,
 6 1, 3 1, 67 1, HERE 176 1, 316 1, 3 1, 47 1, 167 1, 53 1, 5 1,
 3Ø2 1, , RTN :S ↑

CODE /FSTORE ' ALIGN-ADDR COMPILE 345 1, 23 1, 41 1, ' FTEMP
 , 66 1, [illegible] 1, 43 1, 6 1, 3 1, HERE 32 1, 167 1, 257 1, 22 1, 23 1,
  43 1, 5 1, 3Ø2 1, , 62 1, ' FTEMP 12 + , 167 1, 43 1, 353 1, 341 1,
  43 1, 6 1, 3 1, ' HCOPY COMPILE RTN :S ↑

CODE FROUND 17Ø 1, 376 1, 5 1, 334 1, ' ALIGN-ADDR COMPILE
 23 1, 23 1, 26 1, 353 1, 6 1, 3 1, 67 1, HERE 176 1, 316 1, 3 1, 47
  1, 167 1, 53 1, 5 1, 3Ø2 1, , RTN :S ↑

CODE /F/ 16 1, 6 1, HERE 6 1, 377 1, HERE 21 1, ' FTEMP 3 +
 , 41 1, ' FTEMP 7 + , 3Ø5 1, 6 1, 4 1, ' FSUBT COMPILE 3Ø1 1, 4 1,
  322 1, , 72 1, ' FTEMP 12 + , 267 1, 3ØØ 1, 3Ø5 1, 16 1, 3 1,
 ' ALIGN-ADDR COMPILE  :S ↑

353 1, 43 1, 43 1, 43 1, HERE 176 1, 127 1, 346 1, 17 1,
27 1, 27 1, 27 1, 27 1, 262 1, 167 1, 172 1, 346 1, 36Ø 1, 37 1, 37 1,
  37 1, 37 1, 1Ø7 1, 53 1, 15 1, 3Ø2 1, , 3Ø1 1, 21 1, ' FTEMP 3 + ,
  41 1, ' FTEMP 7 + , :S ↑

6 1, 4 1, ' FADDT COMPILE 6 1, 4 1, 41 1, ' FTEMP 3 + , 26 1,
 Ø 1, ' DCRSHIFT COMIILE 15 1, 3Ø2 1, DUP , ' ALIGN-ADDR COMPILE 23 1,
  32 1, 346 1, 36Ø 1, 3Ø2 1, HERE 13 + , 33 1, 32 1, 75 1, 22 1, 16 1,
  1 1, 3Ø3 1, DUP , :S ↑

76 1, 1 1, 62 1, ' FTEMP 12 + , 3Ø3 1, , RTN  CODE FDIV
' CKFACC COMPILE 317 1, ' GETEXP COMOILE 43 1, 176 1, 267 1, 3Ø2 1,
  HERE 5 + , ' FERROR COMPILE 53 1, 171 1, 22Ø 1, 3Ø6 1, 1ØØ 1, 74 1,
  117 1, 27 1, 322 1, HERE 5 + , ' FERROR COMPILE :S ↑

32 1, 261 1, 22 1, ' /FSTORE COMPILE ' /F/ COMPILE
' FROUN COMPILE ' DROP DUP COMPILE COMPILE RTN :S ↑

: FCONSTANT CODE SWAP LITERAL , LITERAL , RTN ;
: FVARIABLE CODE LITERAL, , LITERAL, , RTN ; : F@ [illegible] SWAP @ ; : F! [illegible]
  UNDER SWAP ! 2 UNDER SWAP ! DROP DROP ; : FDROP DROP DROP ;
 :S ↑

 VARIABLE FSAVE CODE FCVRT ' STK-RC COMPILE ' STKDC COMPILE
 353 1, 76 1, 377 1, HERE 74 1, 11 1, JCOP , 6 1, 2 1, 117 1, 355 1,
  ' DC-STK COMPILE ' RC-STK COMPILE RTN :S ↑
: FGOOD [illegible] FCVRT DUP Ø = IF CHARCOUNT @ Ø = IF DROP [illegible]
 [illegible] THEN +CHAR PTRIC @ 1+ PTRIC ! SWAP FSAVE ! POP FSAVE @ ;
: [illegible] Ø CHARCOUNT ! PTRIC ! 2342Ø FGOOD 2342Ø + 175Ø FGOOD
 175Ø + 144 FGOOD 144 + 12 FGOOD 12 + 1 FGOOD DROP FPACK ; :S ↑
: F[illegible] UNDER 2 UNDER ; CODE FSWAP ' ALIGN-ADDR COMPILE
 16 1, 4 1, HERE 1Ø6 1, 32 1, 167 1, 17Ø 1, 22 1, 23 1, 43 1, 15 1,
  3Ø2 1, , RTN :S ↑


: F> FSUB SWAP DROP 1ØØØØØ AND IF Ø ELSE 1 THEN ;
: F< FSWAP F> ; : F= FSUB SWAP DROP 36Ø AND IF Ø ELSE 1 THEN ;
: FUNDER 2* DUP UNDER SWAP 1+ UNDER SWAP ; :S ↑
```

```
F 65535 FCONSTANT FLARGE CODE ISHFT 353 1, 43 1, 43 1, 6 1,
 3 1, ' DCRSHIFT COMPILE RTN CODE FPUT ' POP COMPILE 137 1, 346 1,
  36Ø 1, 37 1, 37 1, 37 1, 37 1, 3Ø6 1, 6Ø 1, 127 1, ' DE-STK COMPILE
  RTN  ;S ↑

: INTCONVERT ØPUSH 1ØØ - DUP HERE 1! 1 DO FPUT ØPUSH HERE
 I + 1! SP STKDE ISHFT LOOP POP POP POP NUMBER ; F Ø FCONSTANT FZERO
  : F>INT FDUP FZERO F< IF FDROP Ø END THEN FDUP FLARGE F> IF FDROP
  177777 END THEN INTCONVERT ;  ;S ↑

: FLOAT DECIMAL F>INT OCTAL ;
```

CONVERS

INTEGER AND FLOATING ARRAY

PACKAGE

GENERAL

The CONVERS array package is designed to allow the user to create both floating point and integer arrays of any arbitrary dimension. To dimension an integer array, the following format is used:

```
A B C . . . X DIMENSION NAME
```

where 'NAME' is the array name. The letters A, B, C, etc. are the scalar integer values of each dimension and 'X' is the number of dimensions of the array. Thus, to dimension an array called 'ARRX', in two dimensions with the scalar value of 10 and 20 in each dimension respectively, the following format is used to create the array:

```
1ϕ 2ϕ 2 DIMENSION ARRX
```

The array 'ARRX' will then have the capability of storing 2ϕϕ integers within itself.

For reasons to be discussed below, 'ARRX' could just as well be defined as a one-dimensional array as follows:

```
2ϕϕ 1 DIMENSION ARRX
```

In this case, 'ARRX' has been defined as a one dimensional array of 2ϕϕ integer locations.

Floating point arrays are created with the 'FDIMENSION' entry with the same format as is used to create integer arrays:

```
A B C . . . X FDIMENSION NAME
```

The values A, B, C, etc. and 'X' are integer values.

Note !!!

1) The CONVERS array source code below was taken off the disk in a block format. Note that the end of each 'block' is followed by the character string ';S ↑'. If the CONVERS array source code is to be 'typed' into the computer (with CONVERS running) or is to be entered on mass storage, the above characters are ignored.

2) The source code shown references a routine called 'JMPEXEC' that is not contained in either the initial machine code dictionary or the standard high level dictionary. (It is in fact defined in our CONVERS disk system.) This was an oversight that was not caught when the CONVERS array package was written or documented. Therefore the user must first define this routine (as follows) before the

CONVERS array package is compiled:

CODE JMPEXEC 3Ø3 1, 237 , RTN

Otherwise, the CONVERS array package is totally compatible with standard 8Ø8Ø CONVERS.

Both integer and floating point arrays are conceptually similar to integer and floating point variables. The only difference is that a displacement value must be on the stack before the array name is executed. For instance, to get the address of the first array element of 'ARRX', the following character string is executed:

1 ARRX

The above character string will replace the 'one' on the stack with the address of the first element of 'ARRX'. The address of the 100th element of 'ARRX' is found by executing the following character string:

1ϕϕ ARRX

Thus, in general, the address of the Nth array element of an array 'NAME' is found by the execution of the generalized character string:

N NAME

The array 'NAME' can be either a floating or integer array.

Since the above generalized character string leaves the address of the Nth array element on the stack, this address can be replaced with the value at this address by the '@' or 'F@' entry depending on whether 'NAME' is an integer or floating point array respectively. Likewise, the Nth array element can be modified by executing the '!' or the 'F!' entry depending again on whether 'NAME' is an integer or floating point array.

Whenever the user attempts to access an array element outside the bounds of the initial dimensioned array value, a fatal error is assumed and the error message '?11' will be entered on the terminal. Whatever entry was executing will terminate, and an unconditional jump will be made to the 'EXECUTIVE' entry. (See note at the beginning of this section). For instance, since 'ARRX' has been dimensioned for 200 values, if the user attempts to access the 201th array element, a fatal error will occur.

EXAMPLES

The following examples will show how an array, once defined, can be treated as an arbitrary array of any dimension. This is because a multi-dimensional array must necessarily be stored in the computer as a single dimensional array. It is up to the user as to the way in which array elements are to be accessed.

Assume that the array, 'NAME' is an integer array dimensioned to 100 decimal. (The user might have created the array as a 10 x 10, 20 x 5, 50 x 2, 5 x 20, 2 x 50, 2 x 5 x 10, 2 x 5 x 2 x 5, etc. The dimension of the array is immaterial since the total array size is 1ϕϕ elements in each case).

Assume that the user wishes to first treat the array as a one dimensional array. As the first example, the user might want to initialize the array with all zeros:

```
: ARRAY-INIT 100 1 DO ϕ I NAME ! LOOP ;
```

To initialize the array 'NAME', the user needs only to execute 'ARRAY-INIT'.

Assume that 'NAME' is now a floating point array. The user wishes to initialize each array element to the value .159 E-13. Since a floating point value can't directly be compiled into a definition, it must first be created as a constant:

```
F .159 E-13 FCONSTANT F*CON
```

The initialization routine may now be written as follows:

```
: ARRAY-INIT 100 1 DO F*CON I NAME F! LOOP ;
```

As a further example, assume that the array is meant to be a two dimensional array with the scalar value of 1ϕ defining each dimension. The user wishes to access the array with the variable 'IROW' representing the 'row' value and 'ICOLUMN' representing the column value. Assume also that the user wishes to define the system such that the column elements are filled first. Thus, each array element must be defined by the proper manipulation of 'ICOLUMN' and 'IROW'. One way of visualizing this problem is subtracting one from the 'row' value, multiply the result by one, and add the required column value. Thus to get the fifth element address in the first row, we subtract by one the row element, leaving a result of zero. this result is multiplied by 1ϕ, leaving again zero. The value '5' is added to the result. Thus we are left with the fifth element in the array. A definition can be written to properly compute the array element address in the following fashion:

```
                 ϕ VARIABLE ICOLUMN
                 ϕ VARIABLE IROW
: ARRAY-ADDR IROW @ 1- 1ϕ * ICOLUMN + @ ;
```

The definition 'ARRAY-ADDR' will compute the proper array address when executed.

The definition 'ARRAY-ADDR' can be used to initialize the two dimensional array 'NAME', to the value of one by defining the following definitions:

```
: INNER 10 1 DO I ICOLUMN ! J IROW ! 1 ARRAY-ADDR NAME ! LOOP ;
: FILL 10 1 DO INNER LOOP ;
```

The definition 'FILL', when executed, would fill the array 'NAME' with the value of one in each array element. It should be easy to see how these examples may be extended to handle any particular situation at hand. Obviously the user has great latitude on how the array should be handled.

ARRAY ROUTINES

* - This routine multiplies two integers on the stack. It is used to compute the amount of memory needed to store a multi-dimensional array.

DIM - This routine computes the amount of array storage needed to define an array.

DIMENSION - This routine is used to define an integer array.

FDIMENSION - This routine is used to define a floating point array.

DIMTEST - This routine tests array for overflow condition.

```
CODE * ' STKDE COMPILE ' STK-BC COMPILE 6 1, Ø 1, 26 1, 11 1,
 HERE 171 1, 37 1, 117 1, 25 1, 3Ø2 1, HERE 6 + , ' BC-STK COMPILE
  311 1, 17Ø 1, 322 1, HERE 3 + , 2Ø3 1, 37 1, 1Ø7 1, 3Ø3 1, , RTN
   :S ↑

: DIM 1 SWAP 1 DO * LOOP 2* ; : DIMTEST < IF END THEN DROP
 77 POP POP OUTTTY 3Ø461 POP OUTTTY POP OUTTTY JMPEXEC ;
 : 4* 2* 2* ;  :S ↑

: DIMENSION CODE ' 2* STK-DICT COMPILE ' DUP STK-DICT COMPILE
 DIM DUP LITERAL , ' DIMTEST STK-DICT COMPILE LITERAL HERE 4 + , ' +
  STK-DICT COMPILE 311 1, HERE + DP ! RTN ;  :S ↑

: FDIMENSION CODE ' 4* STK-DICT COMPILE ' DUP STK-DICT
 COMPILE DIM 2* DUP LITERAL , ' DIMTEST STK-DICT COMPILE LITERAL HERE
  13 + , ' + STK-DICT COMPILE ' DUP STK-DICT COMPILE ' 1+ STK-DICT
  COMPILE ' 1+ STK-DICT COMPILE 311 1, HERE + DP ! RTN ; :S ↑
```

CONVERS

A software system written by Scott B. Tilden and M. Bonner Denton at the Chemistry Department, University of Arizona. Development of this system was partially supported by the Office of Naval Research.

CONVERS DOCUMENTATION AND USERS MANUAL

INTRODUCTION

This manual will be organized around several areas including hardware and software documentation of the CONVERS system as well as a general discussion of the use of the CONVERS software package. During the course of this discussion, many programming examples will be given employing software concepts discussed. In order to get the fullest understanding of CONVERS, it is strongly recommended that the user actually sit at the terminal and experiment using the examples as programming guides. Since CONVERS is interactive in nature, the best way to fully understand CONVERS is in programming with it.

NOTE: CONVERS was originally written for the INTEL® 8080 microcomputer. Systems available in the future will run on the HP® 2100 series and Data General minicomputers.

HARDWARE CONSIDERATIONS

The terminal employed in the authors 8080 system was a teletype (Model 35) that employs a serial current loop. Interfacing to the CPU was through a Processor Technology Inc. 3P + S board that controls serial to parallel conversion between the computer and the teletype. Due to the slow nature of teletype communication vs. the CPU speed, some sort of flag checking is necessary to inform the CPU when data transfer has terminated. Therefore, software I/O routines driving the terminal must be written to first allow terminal communication. If another type of terminal (i.e., a parallel terminal) and/or another interfacing structure is to be used, terminal I/O routines must be rewritten to fit that particular environment.

The CONVERS software to follow uses the following configuration:

- The TTY is defined as I/O port one. (The 8080 allows for 256 ports to be directly addressed.)

- Port zero is defined as the flag (status) port.

The 3P + S uses a UART to control serial to parallel-parallel to serial conversion. Status flags are generated internally to indicate the progress and status of this conversion. Two of these flags, the Data Available Flag (DAV) and the Transmitter Buffer Empty (TBE), are used by the software to determine when data has been received (DAV) and data has been fully transmitted (TBE). The flags from the UART are 'tied' to bits 6 and 7 of the flag (Status) port zero:

- Bit 6_8 is the DAV flag-active high.

- Bit 7_8 is the TBE flag-active high.

Thus, the two terminal I/O routines used are mnemonically as follows: the data to be sent to the terminal is in the 'A' register when entering the 'OUTTTY' routine; data is deposited into the 'A' register upon termination of the 'INTTY' routine:

```
          'OUTTTY'
          PUSH PSW ; Save contents of 'A' register by pushing on stack.
HERE      IN STATUS ; Input current contents of the flag port.  (port Ø)
          ANI 2ϕϕ ; Mask out all bits except bit 7 which is the TBE flag.
          JZ HERE ; Loop to 'HERE' if flag has not been set.
          POP PSW ; Get data on stack.
          OUT DATA ; Output character to terminal.  (port 1)
          RET ; Return to calling program.
```

The above 'OUTTTY' routine starts at address 424_8 and takes up 12 bytes (8 bit words). This routine can be easily rewritten to fit any configuration; 12 bytes should be all the memory space needed.

```
          'INTTY'
HERE1     IN STATUS ; Input current contents of flag port (port ϕ).
          ANI 1ϕϕ$_8$ ; Mask out all bits except bit 6 which is the DAV flag.
          JZ HERE1 ; Loop to 'HERE1' until the DAV flag is set.
          IN DATA ; Input character from port 1.
          ANI 177$_8$ ; Strip parity bit.
          CALL OUTTTY ; Echo character.
          RET ; Return to calling program.
```

The above 'INTTY' routine starts at 2377_8 and is the last routine in the IMCD. If this routine must be rewritten, it can extend 20 to 30 more bytes without <u>any</u> modification to the rest of the program. Note that this routine <u>must</u> strip the parity bit (the MSB) from the ASCII encoded data character. The routine should also call the 'OUTTTY' routine to echo the character that was input if the terminal is operating at full duplex. NOTE: If it is necessary to change only the port addresses or the status bit configuration, this can be done in two ways:

1. Change the EQUATE values STATU (status port), TTY (TTY I/O port), TBE (TBE flag) and DAV (DAV flag) in the source IMCD listing and reassemble.
2. Load the IMCD tape and manually change the following bytes from the S.R. or through a monitor. (Assuming that the starting location of the IMCD has not been changed):

LOCATION	IDENTIFICATION
426_8	Status port.
$43ϕ_8$	Octal value corresponding to Bit location of TBE flag, i.e., bit 7 corresponds to 2ϕϕ octal.
436_8	TTY I/O port.
$24ϕϕ_8$	Status port .

$24\phi2_8$	Octal value corresponding to bit location of DAV flag, i.e. bit 6 corresponds to 1ϕϕ octal.
$24\phi7_8$	TTY I/O port.

CONFIGURATING FOR ANOTHER STARTING LOCATION

The CONVERS software system as supplied is configured for a starting location of $1\phi\phi_8$. To move the system up in memory will require some effort. However, note that only the IMCD must be recompiled (reconfigured) since the rest of the CONVERS system is supplied as a SOURCE tape which the IMCD itself will compile and load into memory.

To move the IMCD up in memory, a source tape of the IMCD listing (see the end of this section) should be made reassembled for another memory location. The IMCD source listing shown is compatible with an INTEL® assembler.

CONFIGURING FOR ANOTHER MEMORY SIZE

The IMCD as supplied is configured for a 12K memory size. This can be changed as follows:

1. Change the EQUATE value of POP? from 61_8 to the value corresponding to the 8 MSB's of the memory bound plus two. For instance, to configure for 8K, POP? would be re-equated with 41_8.

The label value for SPA must also be changed to correspond to the memory bound plus one (16 bits). For instance, to configure for 8K, SPA would be redefined to equal 20000_8. The IMCD source is then reassembled.

2. The following locations can be changed manually (assuming the IMCD has not been reconfigured for another address) through the SR or a monitor:

ADDRESS	CHANGE TO
333_8	Value of POP? (see 1 above)
$3\phi7$ & $31\phi_8$	Value of SPA (16 bits) (see SPA in 1 above)

LOADING THE IMCD INTO MEMORY

The IMCD tape supplied is in 3-digit octal format. It should be loaded starting at $1\phi\phi_8$. The following routine can be used as the loader:

```
          DAV   EQU - DAV FLAG BIT VALUE
          STATU EQU - STATUS PORT
          DATA  EQU - TTY I/O PORT
                LXI D, ADDR ; LOAD STARTING
NUM             MVI L,Ø     ; ADDRESS - CONVERS
INTTY           IN STATU    ; STARTING ADDRESS
                ANI DAV     ; IS 1ØØ OCTAL
                JZ INTTY
                IN DATA
                OUT DATA
                ANI 177_8
                SBI Ø6Ø_8       ; SUB. ASCII OFFSET
                JM ONEWD        ; IS IT A NUMBER?
                DAD H           ; ADD TO 'L' REG
                DAD H
                DAD H
                ADD L
                MOV L,A
                JMP INTTY       ; GET NEXT CHAR
ONEWD           CPI 36Ø_8       ; IS IT A SPACE
                JNZ NUM         ; NO-TRY AGAIN
                MOV A,L         ; STORE NUMBER
                STAX D
                INX D
                JMP NUM
```

DICTIONARY ENTRY DOCUMENTATION - Initial Machine Code Dictionary (IMCD)

The following discussion will document the IMCD routines in both description and flow chart form. This discussion will treat each routine as an entity; an overview of the system will come later.

UPDICT

UPDICT is an entry that inputs characters from the terminal and deposits them sequentially into the dictionary. In the first location will be deposited the total number of inputted characters. A space $(4\phi_8)$ is the terminating character; i.e., characters will be inputted until a space is detected. The starting location for depositing characters is found at the variable location DP which is the first two bytes of the IMCD. Since DP always points to the last datum to be deposited in the dictionary, UPDICT first increments DP (in H & L reg.). UPDICT itself does not change the value of DP. UPDICT ignores all characters with ASCII values less than octal 16. Note that UPDICT first clears the next four locations in the dictionary. A maximum of 128_{10} characters may be entered.

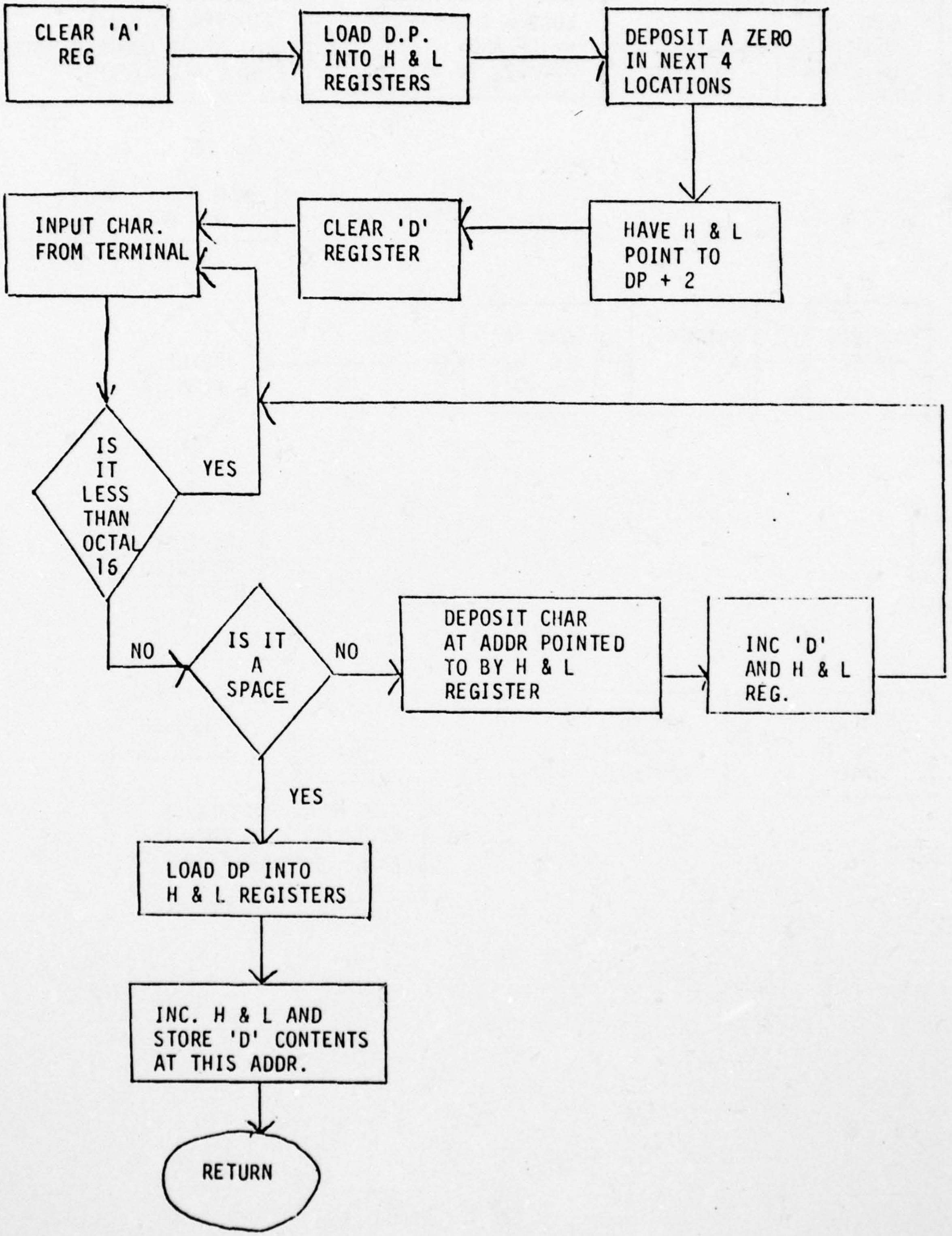
DUMMY

DUMMY is a 'dummy' routine which can be anything the user wishes it to be. DUMMY initially will call the NULL routine, however, by changing the destination address of this CALL, DUMMY will execute any routine.

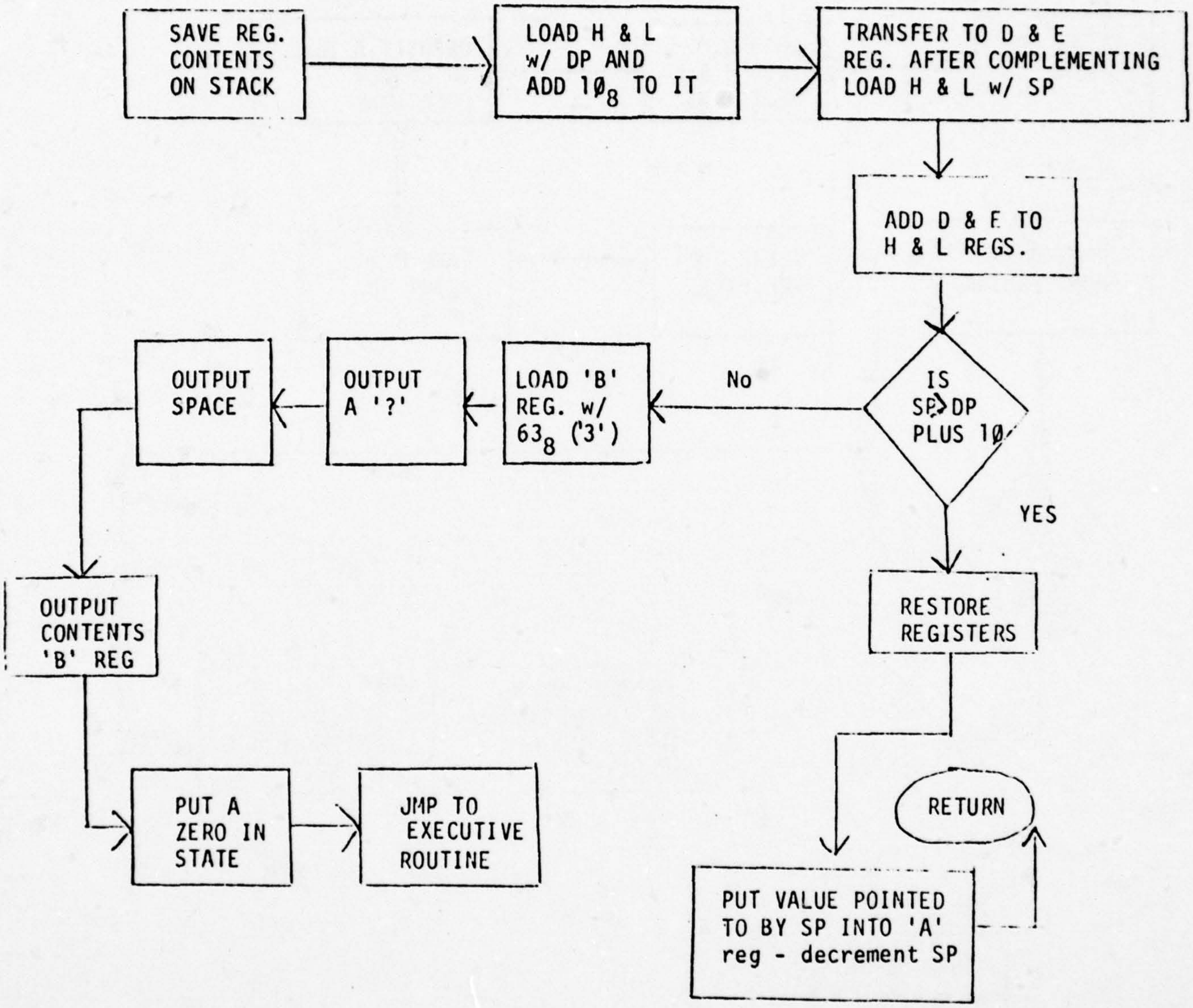
PUSH

PUSH is a routine that in its simplest form puts the contents of the 'A' register onto the software stack. It also decrements the stack pointer. (The software stack extends towards low memory.) Initially, PUSH also checks for stack overflow, i.e., is the stack about to over-run the dictionary? To perform this test PUSH first adds 1Ø$_8$ to the current DP.

UPDICT FLOW CHART


CLEAR 'A' REG
LOAD D.P. INTO H & L REGISTERS
DEPOSIT A ZERO IN NEXT 4 LOCATIONS
HAVE H & L POINT TO DP + 2
CLEAR 'D' REGISTER
INPUT CHAR. FROM TERMINAL
IS IT LESS THAN OCTAL 16
YES
NO
IS IT A SPACE
NO
DEPOSIT CHAR AT ADDR POINTED TO BY H & L REGISTER
INC 'D' AND H & L REG.
YES
LOAD DP INTO H & L REGISTERS
INC. H & L AND STORE 'D' CONTENTS AT THIS ADDR.
RETURN

PUSH FLOW CHART

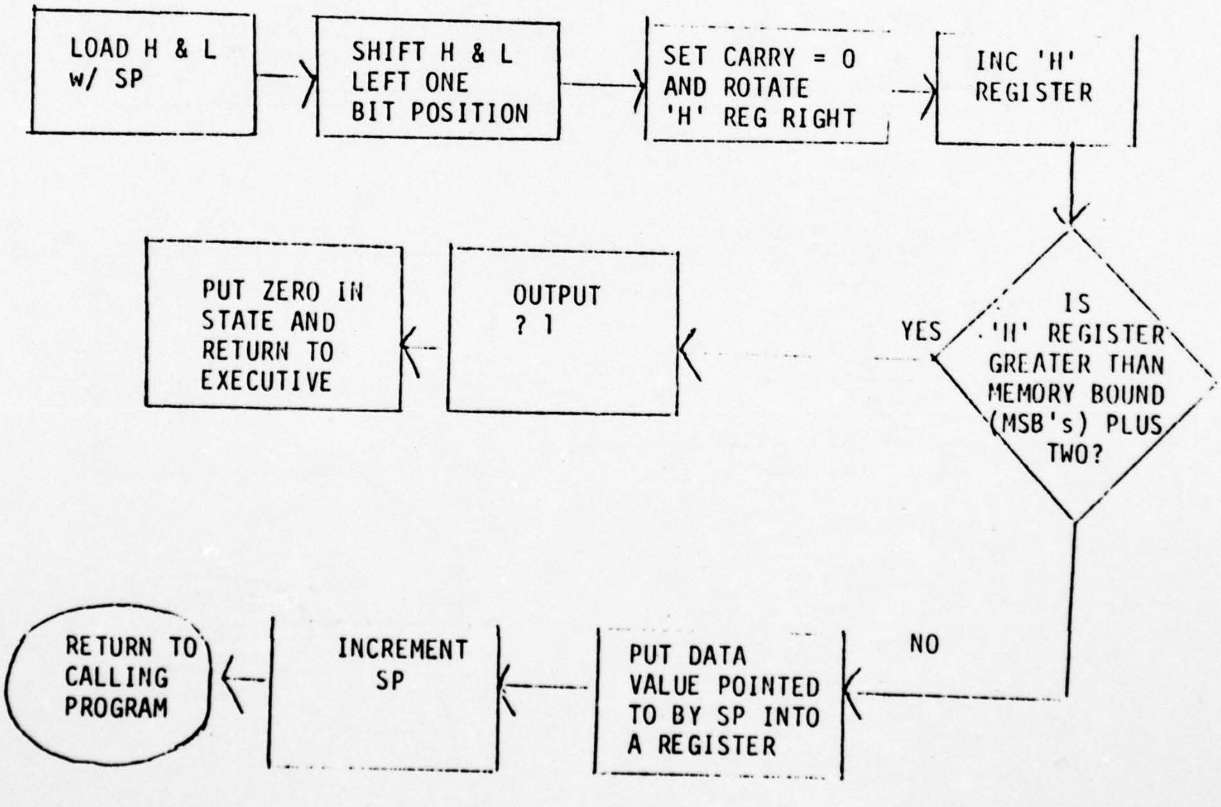

SAVE REG. CONTENTS ON STACK
LOAD H & L w/ DP AND ADD 1Ø$_8$ TO IT
TRANSFER TO D & E REG. AFTER COMPLEMENTING LOAD H & L w/ SP
ADD D & E TO H & L REGS.
IS SP>DP PLUS 1Ø
No
YES
LOAD 'B' REG. w/ 63$_8$ ('3')
OUTPUT A '?'
OUTPUT SPACE
OUTPUT CONTENTS 'B' REG
PUT A ZERO IN STATE
JMP TO EXECUTIVE ROUTINE
RESTORE REGISTERS
PUT VALUE POINTED TO BY SP INTO 'A' reg - decrement SP
RETURN

(This allows one to recover the system by manually resetting either or both the stack pointer or the dictionary pointer.) It then compares this value with the current stack pointer, if DP (+10) is now greater than SP an error message is outputted (?1). This error routine also sets STATE to zero and unconditionally jumps to the EXECUTE routine. Under some conditions stack overflow checks are not necessary; (i.e., when using developed programs in a fast environment) overflow checking can be bypassed by incrementing the CODE POINTER of PUSH by three. CODE POINTER incrementing is usually done under software control. Only the 'A', 'H' and 'L' registers are effected by PUSH.

POP

POP is an entry that again in its simplest form puts the number pointed to by the SP into the A register and increments the SP. Initially stack underflow error checking is done, this can be eliminated under software control by adding 23_8 to the code pointer of the entry. Only the 'A', 'H & L' registers are effected.

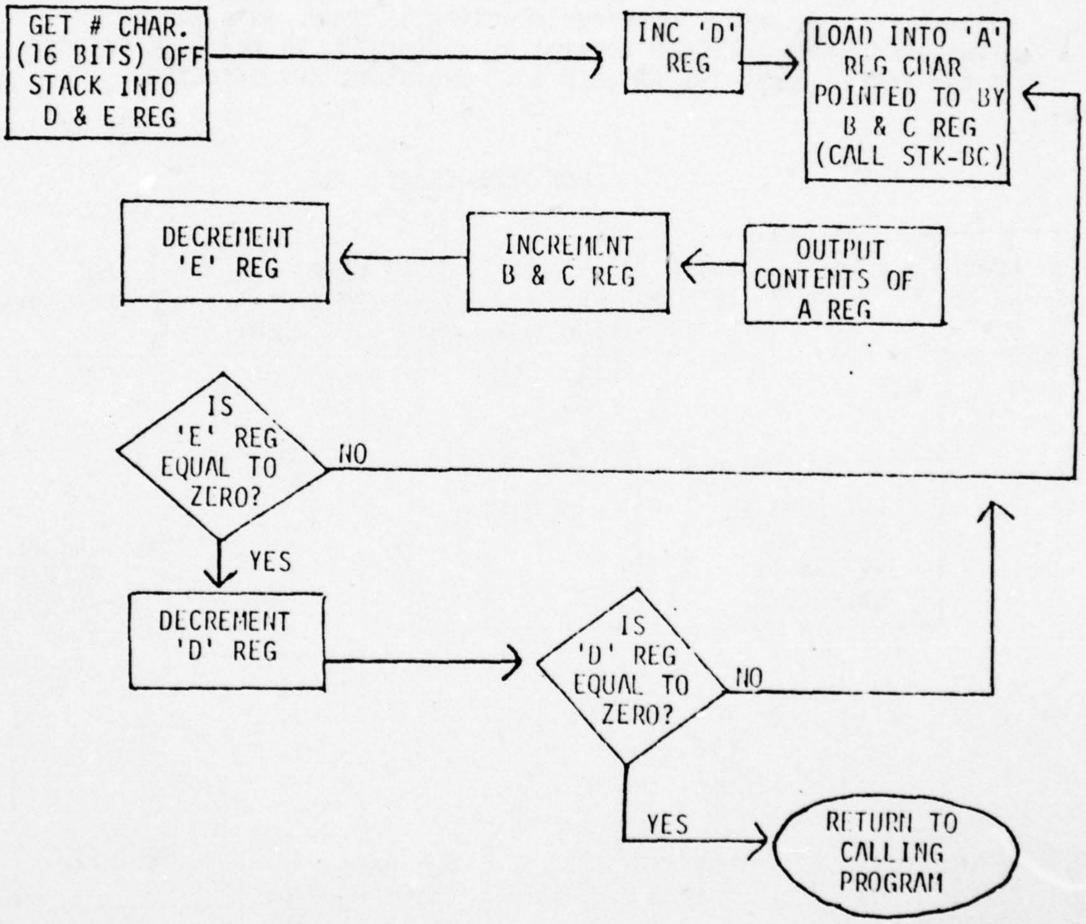
POP FLOW CHART


LOAD H & L w/ SP
SHIFT H & L LEFT ONE BIT POSITION
SET CARRY = 0 AND ROTATE 'H' REG RIGHT
INC 'H' REGISTER
IS 'H' REGISTER GREATER THAN MEMORY BOUND (MSB's) PLUS TWO?
YES
OUTPUT ? 1
PUT ZERO IN STATE AND RETURN TO EXECUTIVE
NO
PUT DATA VALUE POINTED TO BY SP INTO A REGISTER
INCREMENT SP
RETURN TO CALLING PROGRAM

TTYOUT

TTYOUT will output a block of characters to the terminal. TTYOUT should find the number of the characters on top of the stack and the starting address of the character block underneath the top value. (A number on the stack refers to a 16 bit number;)

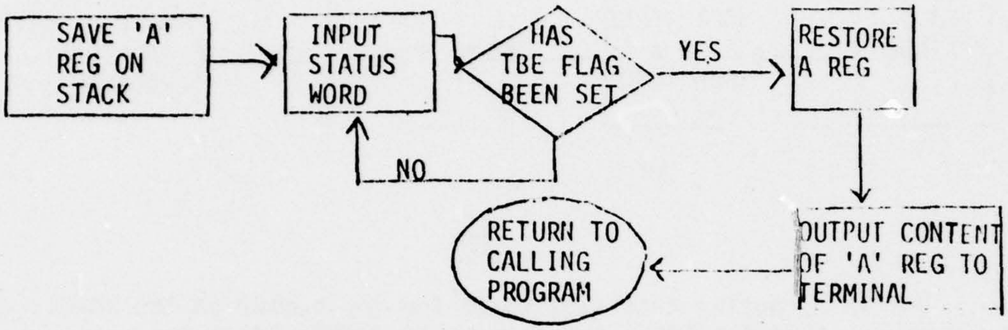
TTYOUT FLOW CHART


GET # CHAR. (16 BITS) OFF STACK INTO D & E REG
INC 'D' REG
LOAD INTO 'A' REG CHAR POINTED TO BY B & C REG (CALL STK-BC)
OUTPUT CONTENTS OF A REG
INCREMENT B & C REG
DECREMENT 'E' REG
IS 'E' REG EQUAL TO ZERO?
NO
YES
DECREMENT 'D' REG
IS 'D' REG EQUAL TO ZERO?
NO
YES
RETURN TO CALLING PROGRAM

OUTTTY

OUTTTY will output the contents of the 'A' register to the terminal. Assuming a serial UART data transfer:

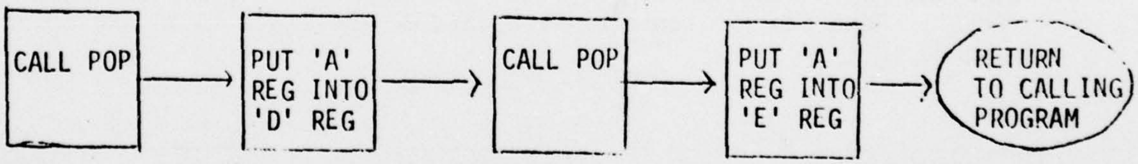
OUTTTY FLOW CHART



STKDE

STKDE loads the D & E registers with the contents on top of the stack. It does this by calling POP twice, the first time putting the 'A' register into the 'D' register, the next time the 'A' register is put into the 'E' register.

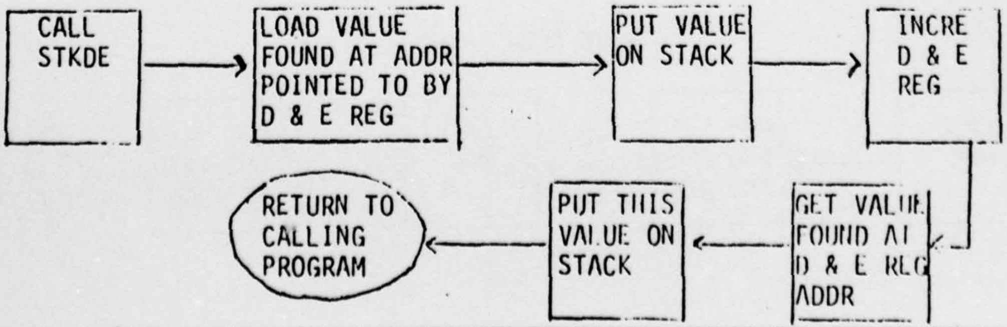
STKDE FLOW CHART



@

The routine '@' replaces the number on the stack with the 16 bit number contained at this address.

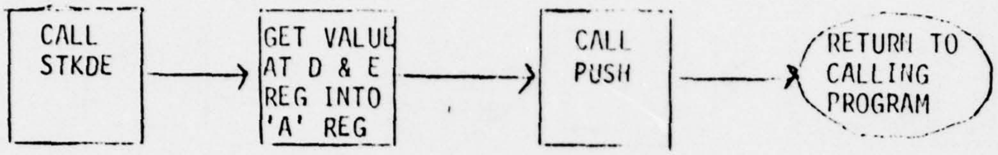
@ FLOW CHART


CALL STKDE
LOAD VALUE FOUND AT ADDR POINTED TO BY D & E REG
PUT VALUE ON STACK
INCRE D & E REG
GET VALUE FOUND AT D & E REG ADDR
PUT THIS VALUE ON STACK
RETURN TO CALLING PROGRAM

1@

The routine '1@' replaces the number on the stack with the 8-bit value contained at this address.

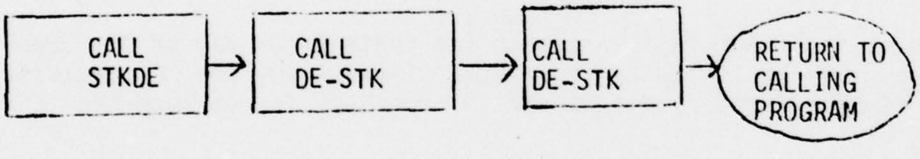
1@ FLOW CHART


CALL STKDE
GET VALUE AT D & E REG INTO 'A' REG
CALL PUSH
RETURN TO CALLING PROGRAM

DUP

DUP is a routine that duplicates the top number on the stack. This is done by calling 'STKDE' and then calling 'DE-STK' twice.

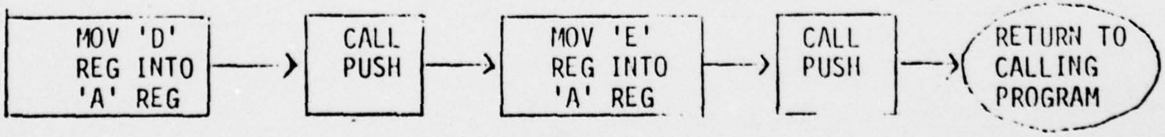
DUP FLOW CHART



DE-STK

DE-STK pushes the contents of the D & E registers on the stack. It does this by moving the 'D' register into the 'A' register and calling 'PUSH'. Next the 'E' register is pushed on the stack in an analogous manner.

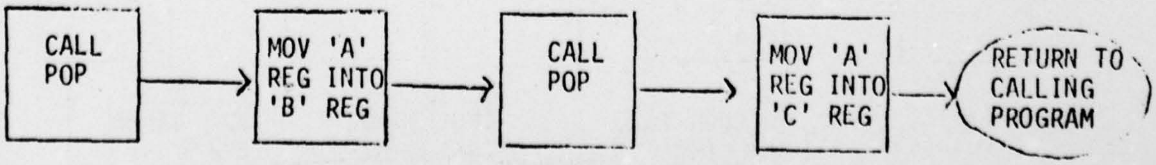
DE-STK FLOW CHART



STK-BC

STK-BC pops the top number on the stack and puts into the B & C registers.

STK-BC FLOW CHART



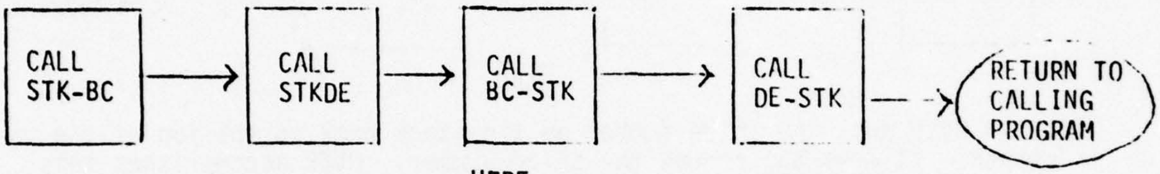
BC-STK

BC-STK pushes onto the stack the contents of the B & C registers. This is done in an analogous fashion to the 'DE-STK' routine. (See DE-STK)

SWAP

SWAP simply swaps the top two numbers on the stack. This is done by calling 'STK-BC' and STKDE which places the top two numbers into the B & C and D & E registers. It then calls BC-STK and DE-STK in that order which effectively swaps the order of the two numbers on the stack.

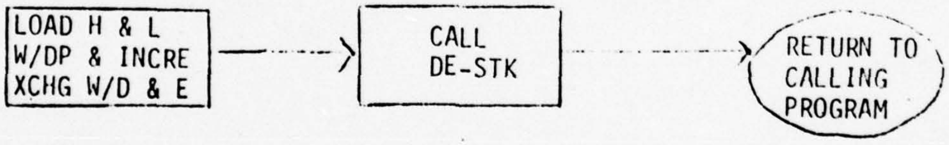
SWAP FLOW CHART



HERE

HERE puts the current DP+1 on the stack. Therefore 'HERE' will supply the address of the next available location in the dictionary at that time. 'HERE' is used quite often when compiling 'CODE' definitions for compiling destination addresses of JMP's, etc. inside the definition. 'HERE' supplies DP+1 by loading the contents of the H & L registers with DP, exchanging with the D & E registers and then calling 'DE-STK'.

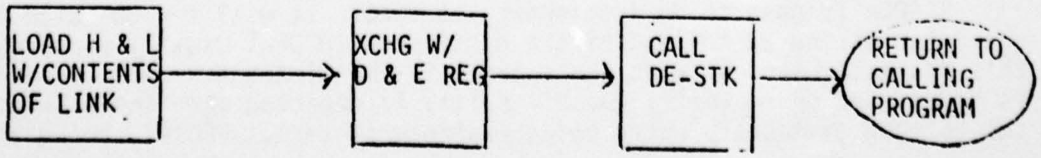
HERE FLOW CHART


LOAD H & L W/DP & INCRE XCHG W/D & E
CALL DE-STK
RETURN TO CALLING PROGRAM

HEAD

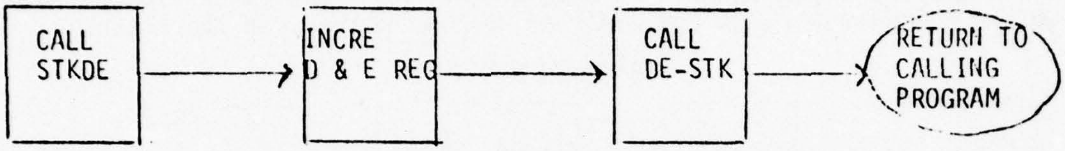
HEAD puts on the stack the contents of the LINK location (LINK is contained in the third and fourth byte of the IMCD) and puts it on the stack. LINK always points to the first character of the last dictionary entry to be completely compiled. Note that when compiling a new entry in the dictionary LINK does not point to the first character of this new entry. Only upon ending the entry will LINK subsequently be updated.

HEAD FLOW CHART



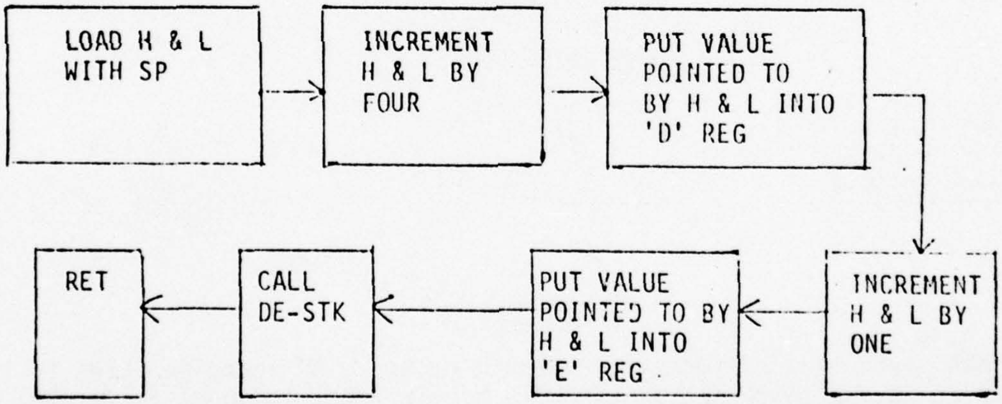
1+

The definition '1+' simply increments the number on top of the stack by one. This is done by calling 'STKDE', incrementing the D & E registers by one then calling 'DE-STK' to put the incremented number back on the stack. Some high level definitions later on will allow one to add numbers on the stack, however, in many cases, the number only needs to be incremented by one in which case '1+' will do it much faster.



OVER

OVER puts the third number on the stack back on the top of the stack. It does not remove the third number. OVER accomplishes this function by loading the current stack pointer contents into the H & L registers. This is incremented by four. The H & L registers now point the MSB of the third number on the stack, the value at this address is copied into the 'D' register. The H & L registers are incremented once more and this value (the LSB of the third number on the stack) is copied into the 'E' register. DE-STK is called to push the copy of the third number under the stack back on top of the stack.


LOAD H & L WITH SP
INCREMENT H & L BY FOUR
PUT VALUE POINTED TO BY H & L INTO 'D' REG
INCREMENT H & L BY ONE
PUT VALUE POINTED TO BY H & L INTO 'E' REG
CALL DE-STK
RET

DROP

DROP removes the top number from the stack. It does this by calling 'POP' twice.

SEARCH

SEARCH is used to do dictionary searches. It will try to match the character string at the end of the dictionary (UPDICT usually supplies this character string) with the entries in the dictionary until a match is found. No match (entry doesn't exist) is detected when the LINK of the current dictionary entry being examined is zero. UPDICT, being the first entry in the dictionary, has its LINK set to zero, therefore, dictionary searches will terminate here.

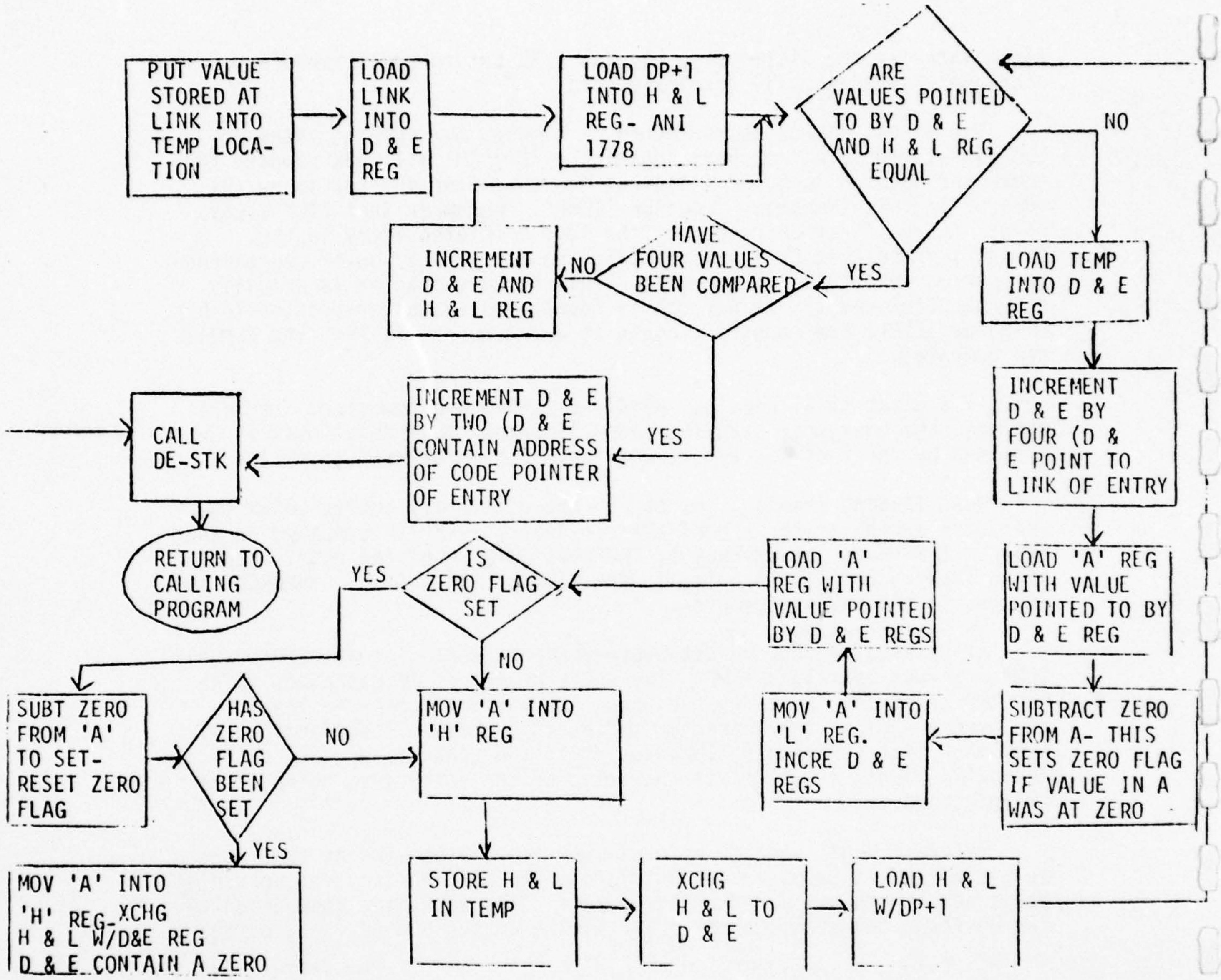
The actual searching procedure is simple, the current value of LINK is stored in a temporary location. 'SEARCH' will now compare the character pointed to by DP+1 against the character pointed to by the address in this temporary location 'TEMP'. Remember that LINK always points to the first character of the last completed entry in the dictionary and that DP+1 will contain the first location of the character string that 'UPDICT' supplies. This first character is actually the character count. If a match is found, the character pointed to by DP+2 and TEMP+1 are compared, again if a match results DP+3 and TEMP+2 are compared.

If a match still results, DP+4 and TEMP+3 are compared. If this matches, the entry has been located. (Remember that uniqueness is determined by the # of characters and the first three characters.)

Upon finding a match, ie, one of the dictionary entrys match the character string at the end of the dictionary, TEMP+6 is pushed on the stack. TEMP+6 is the address of the CODE POINTER of the entry found above (TEMP+4 and TEMP+5 point to the LINK of the entry). 'SEARCH' now returns to the calling program.

If a match did not result above the word contained at TEMP+4 and TEMP+5 is put back into TEMP. The above procedure is repeated, which has the effect of comparing the second to the last entry in the dictionary against the character string at the end of the dictionary. (Remember that TEMP+4 and TEMP+5 point to the LINK of the last entry. This LINK points to the first character of the entry just below it in the dictionary.)

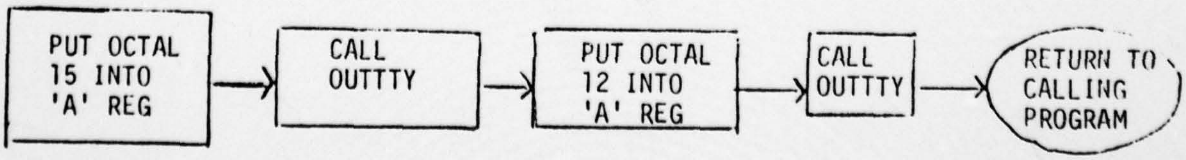
Before TEMP is updated as explained above, the LINK of the entry just examined is compared against zero. If the LINK is zero, searching stops and a zero is placed on the stack. This indicates that no entry can be found to match the character string at the end of the dictionary.


PUT VALUE STORED AT LINK INTO TEMP LOCATION
LOAD LINK INTO D & E REG
LOAD DP+1 INTO H & L REG- ANI 1778
ARE VALUES POINTED TO BY D & E AND H & L REG EQUAL
NO
YES
HAVE FOUR VALUES BEEN COMPARED
NO
INCREMENT D & E AND H & L REG
YES
INCREMENT D & E BY TWO (D & E CONTAIN ADDRESS OF CODE POINTER OF ENTRY
CALL DE-STK
RETURN TO CALLING PROGRAM
LOAD TEMP INTO D & E REG
INCREMENT D & E BY FOUR (D & E POINT TO LINK OF ENTRY
LOAD 'A' REG WITH VALUE POINTED TO BY D & E REG
SUBTRACT ZERO FROM A- THIS SETS ZERO FLAG IF VALUE IN A WAS AT ZERO
MOV 'A' INTO 'L' REG. INCRE D & E REGS
LOAD 'A' REG WITH VALUE POINTED BY D & E REGS
IS ZERO FLAG SET
YES
NO
SUBT ZERO FROM 'A' TO SET-RESET ZERO FLAG
HAS ZERO FLAG BEEN SET
NO
YES
MOV 'A' INTO 'H' REG
MOV 'A' INTO 'H' REG-XCHG H & L W/D&E REG D & E CONTAIN A ZERO
STORE H & L IN TEMP
XCHG H & L TO D & E
LOAD H & L W/DP+1

CRLF

CRLF outputs a CR and LF to the terminal. It does this by loading into the 'A' register the respective ASCII code and calling the 'OUTTY' routine.

CRLF FLOW CHART


PUT OCTAL 15 INTO 'A' REG
CALL OUTTTY
PUT OCTAL 12 INTO 'A' REG
CALL OUTTTY
RETURN TO CALLING PROGRAM

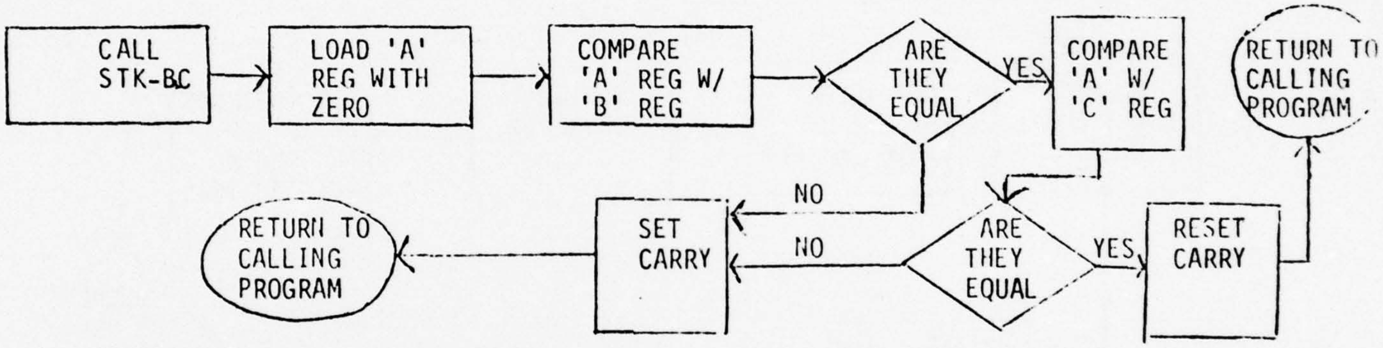
SPACE

SPACE outputs a space (octal 4∅) to the terminal in an analogous procedure as 'CRLF'.

ZERO?

ZERO? tests the number on the stack for a zero. If zero, the carry flag is reset, if not zero, the carry flag is set.

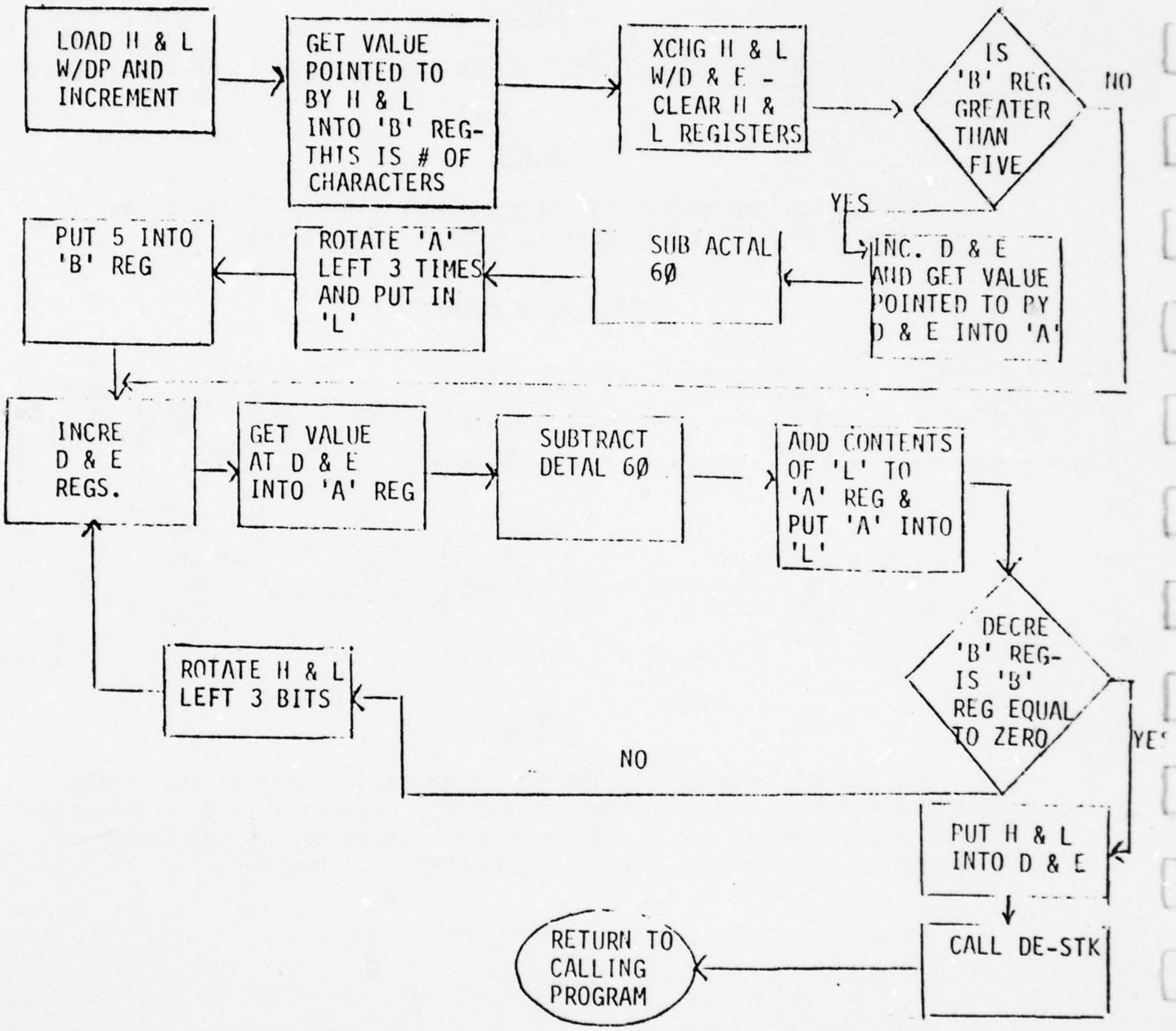
ZERO? FLOW CHART



OCTCVRT

OCTCVRT will convert the string of characters at the end of the dictionary into a binary number. "OCTCVRT' assumes that this character string all represent ASCII octal numbers. There can be any number of characters, however only the first six will be converted.

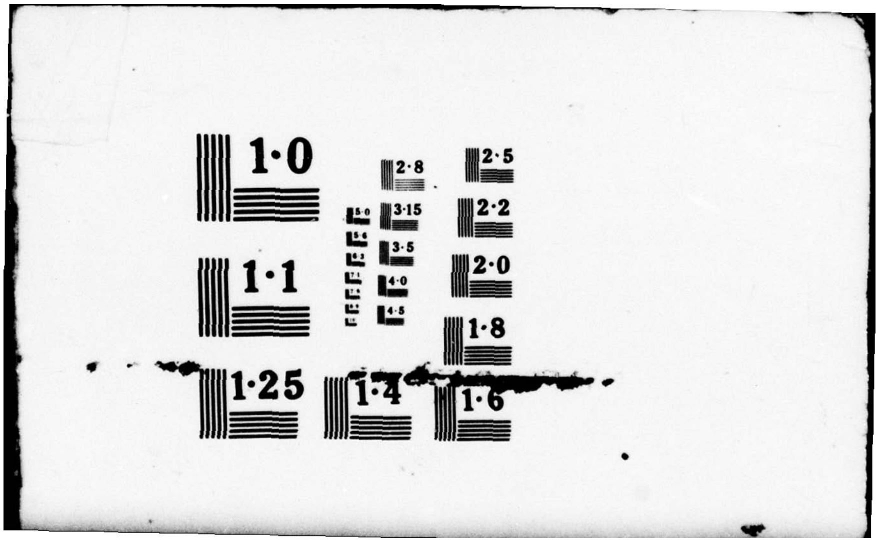
OCTCVRT FLOW CHART


LOAD H & L W/DP AND INCREMENT
GET VALUE POINTED TO BY H & L INTO 'B' REG-THIS IS # OF CHARACTERS
XCHG H & L W/D & E - CLEAR H & L REGISTERS
IS 'B' REG GREATER THAN FIVE
NO
YES
INC. D & E AND GET VALUE POINTED TO BY D & E INTO 'A'
SUB ACTAL 6Ø
ROTATE 'A' LEFT 3 TIMES AND PUT IN 'L'
PUT 5 INTO 'B' REG
INCRE D & E REGS.
GET VALUE AT D & E INTO 'A' REG
SUBTRACT DETAL 6Ø
ADD CONTENTS OF 'L' TO 'A' REG & PUT 'A' INTO 'L'
DECRE 'B' REG-IS 'B' REG EQUAL TO ZERO
YES
NO
ROTATE H & L LEFT 3 BITS
PUT H & L INTO D & E
CALL DE-STK
RETURN TO CALLING PROGRAM

AD-A067 746 ARIZONA UNIV TUCSON DEPT OF CHEMISTRY F/G 9/2
CONVERS: AN INTERPRETIVE COMPILER.(U)
APR 79 S B TILDEN, M B DENTON N00014-75-C-0513
UNCLASSIFIED TR-19 NL

2 OF 2
ADA
067746

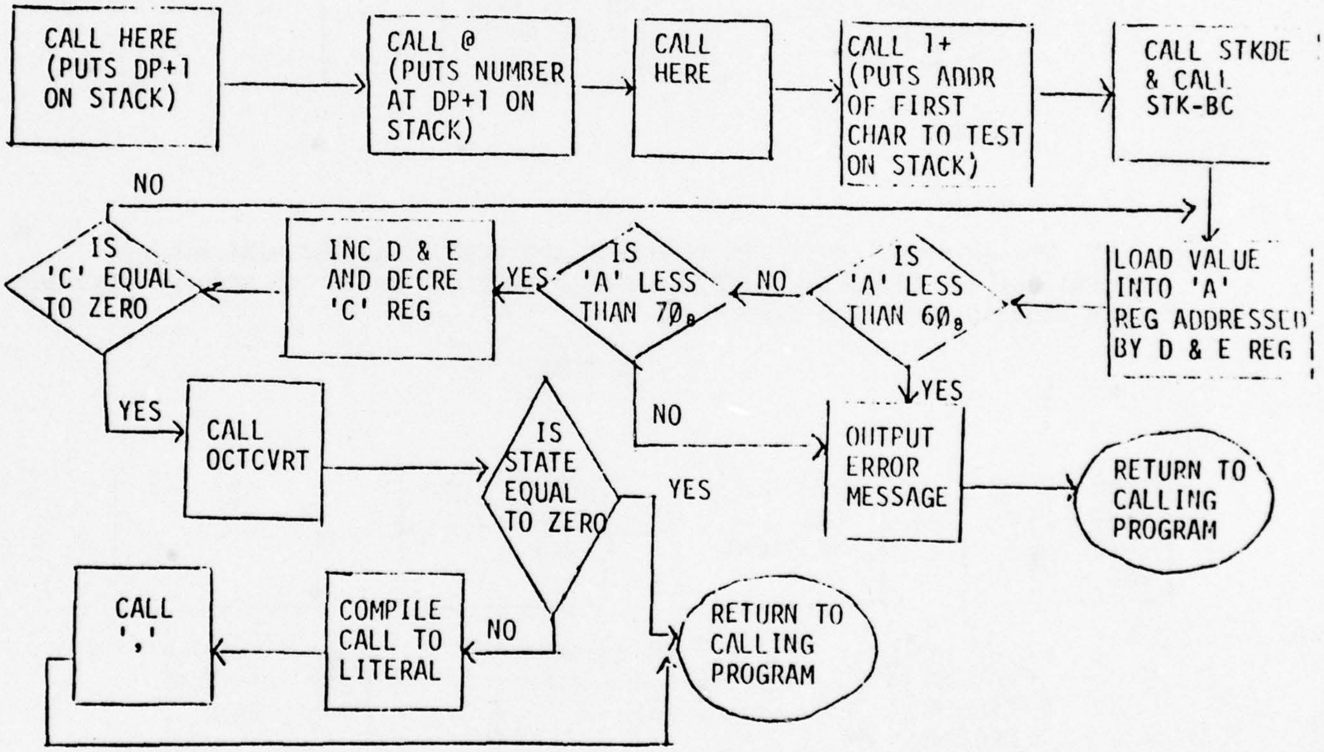
END
DATE
FILMED
6-79
DDC


1·0
2·8
2·5
5·0
3·15
2·2
3·5
2·0
4·0
1·1
4·5
1·8
1·25
1·4
1·6

NUMBER

NUMBER checks the character string at the end of the dictionary to see if all the characters can be interpreted as octal (or decimal) numbers. If not, an error message is outputted. If the character string can be interpreted as a number, NUMBER calls 'OCTCVRT' whereupon the character string is converted and put on the stack. 'NUMBER' then checks the current value of the variable STATE, if STATE is a zero, NUMBER does nothing else and returns to the calling program. Otherwise, a call to LITERAL is compiled into the dictionary along with the number on the stack. STATE will not equal zero when compiling colon (:) definitions.

NUMBER FLOW CHART

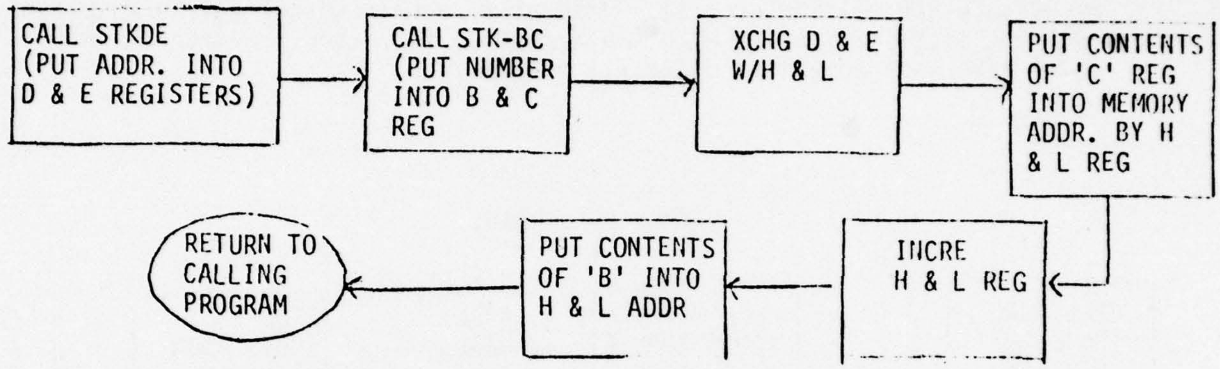


CALL HERE (PUTS DP+1 ON STACK)
CALL @ (PUTS NUMBER AT DP+1 ON STACK)
CALL HERE
CALL 1+ (PUTS ADDR OF FIRST CHAR TO TEST ON STACK)
CALL STKDE & CALL STK-BC
NO
LOAD VALUE INTO 'A' REG ADDRESSED BY D & E REG
IS 'A' LESS THAN 60₈
NO
IS 'A' LESS THAN 70₈
YES
INC D & E AND DECRE 'C' REG
IS 'C' EQUAL TO ZERO
YES
YES
NO
CALL OCTCVRT
IS STATE EQUAL TO ZERO
YES
OUTPUT ERROR MESSAGE
RETURN TO CALLING PROGRAM
NO
COMPILE CALL TO LITERAL
CALL ','
RETURN TO CALLING PROGRAM

!

The entry ! puts the number under the stack into the address on top of the stack.

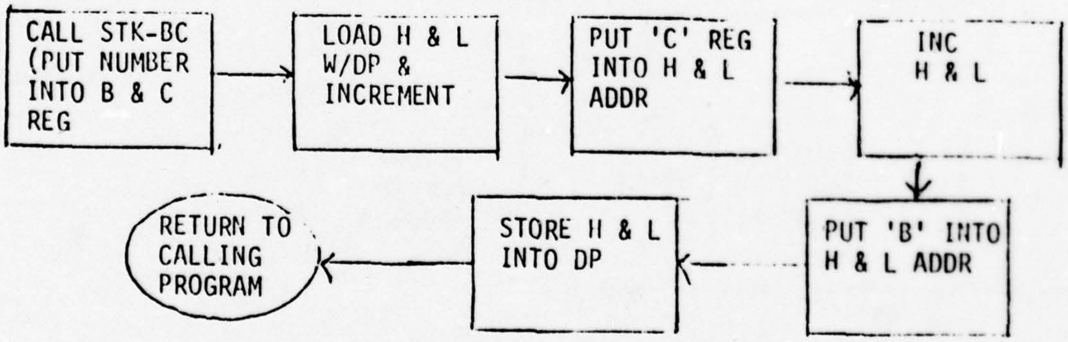
! FLOW CHART


CALL STKDE (PUT ADDR. INTO D & E REGISTERS)
CALL STK-BC (PUT NUMBER INTO B & C REG
XCHG D & E W/H & L
PUT CONTENTS OF 'C' REG INTO MEMORY ADDR. BY H & L REG
INCRE H & L REG
PUT CONTENTS OF 'B' INTO H & L ADDR
RETURN TO CALLING PROGRAM

,

The entry ',' puts the number on the stack into the next memory location(s). Since the 8080 works with an 8 bit word, two memory locations are used to store the number.

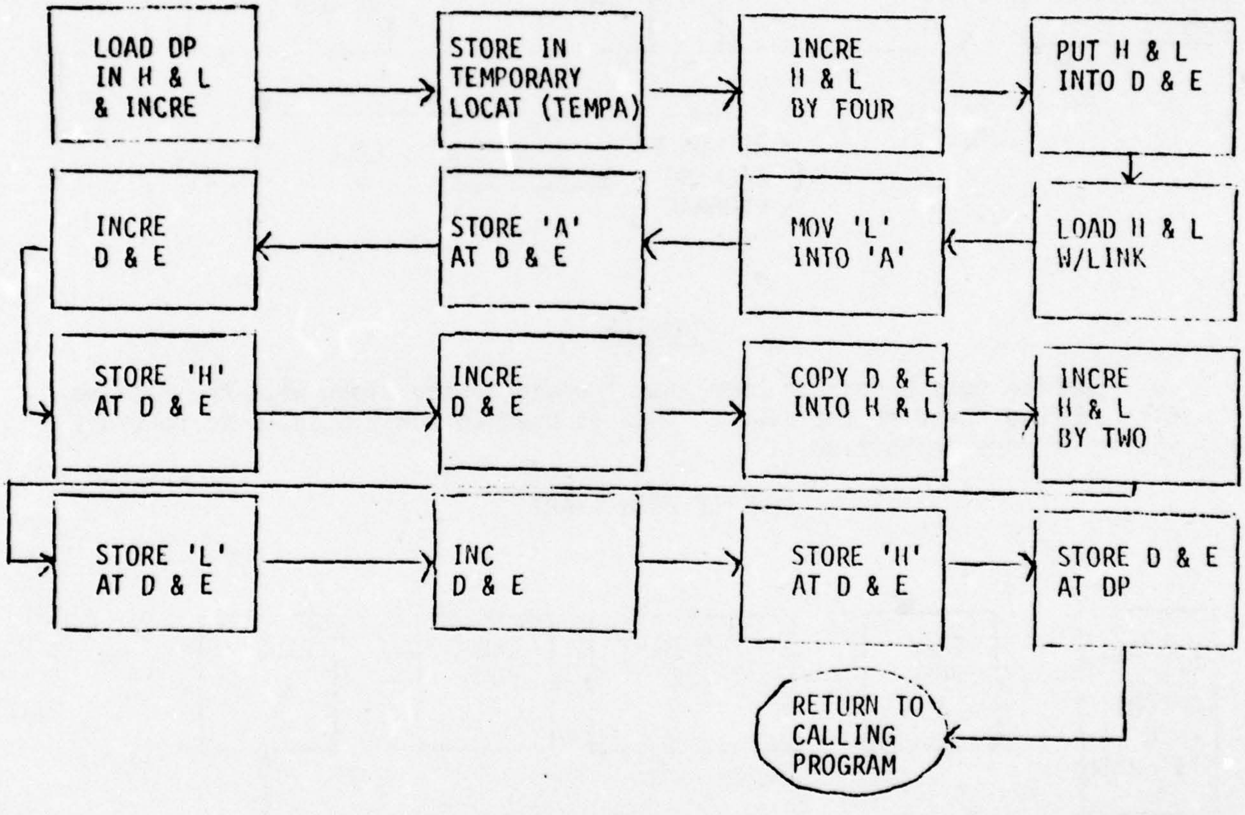
, FLOW CHART


CALL STK-BC (PUT NUMBER INTO B & C REG
LOAD H & L W/DP & INCREMENT
PUT 'C' REG INTO H & L ADDR
INC H & L
PUT 'B' INTO H & L ADDR
STORE H & L INTO DP
RETURN TO CALLING PROGRAM

ENTER

ENTER is used by all definitions that create new definitions (CODE and ':'). ENTER stores the current DP+1 in a temporary location (TEMPA). It then stores the current LINK in DP+5 and DP+6 which effectively compiles the LINK of the new entry. It then puts DP+9 into the location addressed by DP+7 and DP+8. This effectively compiles the CODE POINTER of the entry to point to the next location after the CODE POINTER. It finally stores DP+8 into the DP which updates DP to the next available location in the dictionary.

ENTER FLOW CHART



LINK

After a new dictionary entry has been completed, the way the entry is officially made part of the system is to update the variable LINK to point to the first location of the new entry. This is done by storing the value of TEMPA (which ENTRY has supplied) into the variable LINK.

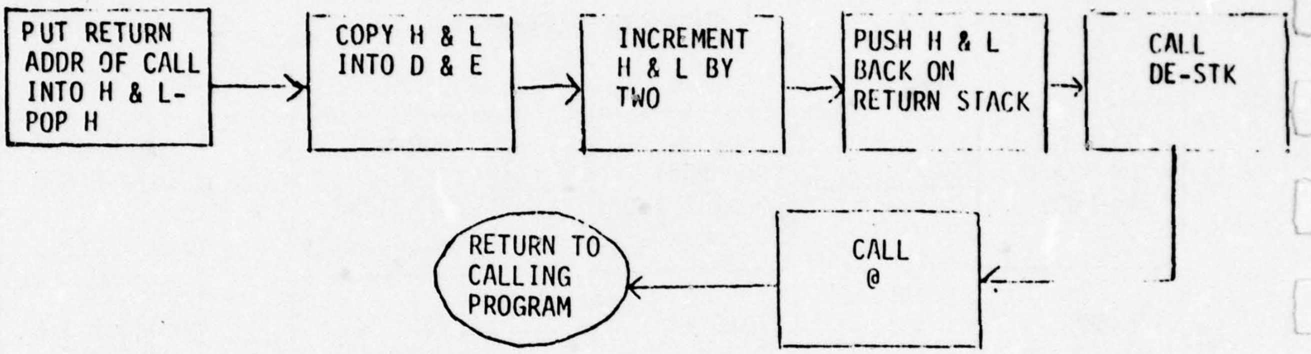
1,

The entry '1,' stores the eight LSB's of the number on the stack into the next available dictionary location. This is accomplished much like ',' except only the 'C' reg contents are put into the dictionary. (See ','.)

LITERAL

LITERAL has two parts-one part simply puts a call to the starting address of Part 2 into the dictionary and returns. PART-2, upon execution, will place the value of the contents of the next two bytes after this call onto the stack. Thus, LITERAL is used to push constants inside of entries onto the stack upon execution of the entry.

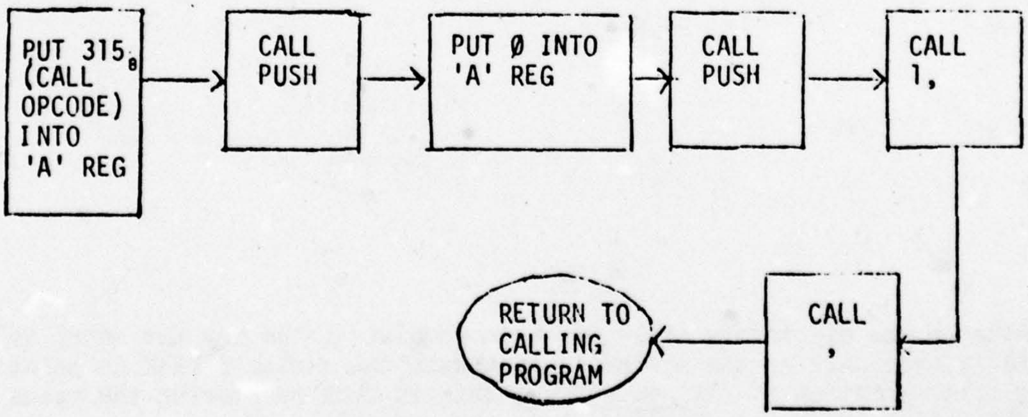
LITERAL FLOW CHART-PART 2


PUT RETURN ADDR OF CALL INTO H & L-POP H
COPY H & L INTO D & E
INCREMENT H & L BY TWO
PUSH H & L BACK ON RETURN STACK
CALL DE-STK
CALL @
RETURN TO CALLING PROGRAM

COMPILE

COMPILE puts a call op code into the dictionary along with the destination address found on the stack. This is used to compile calls to routines inside of code definitions.

COMPILE FLOW CHART


PUT 315_8 (CALL OPCODE) INTO 'A' REG
CALL PUSH
PUT Ø INTO 'A' REG
CALL PUSH
CALL 1,
CALL ,
RETURN TO CALLING PROGRAM

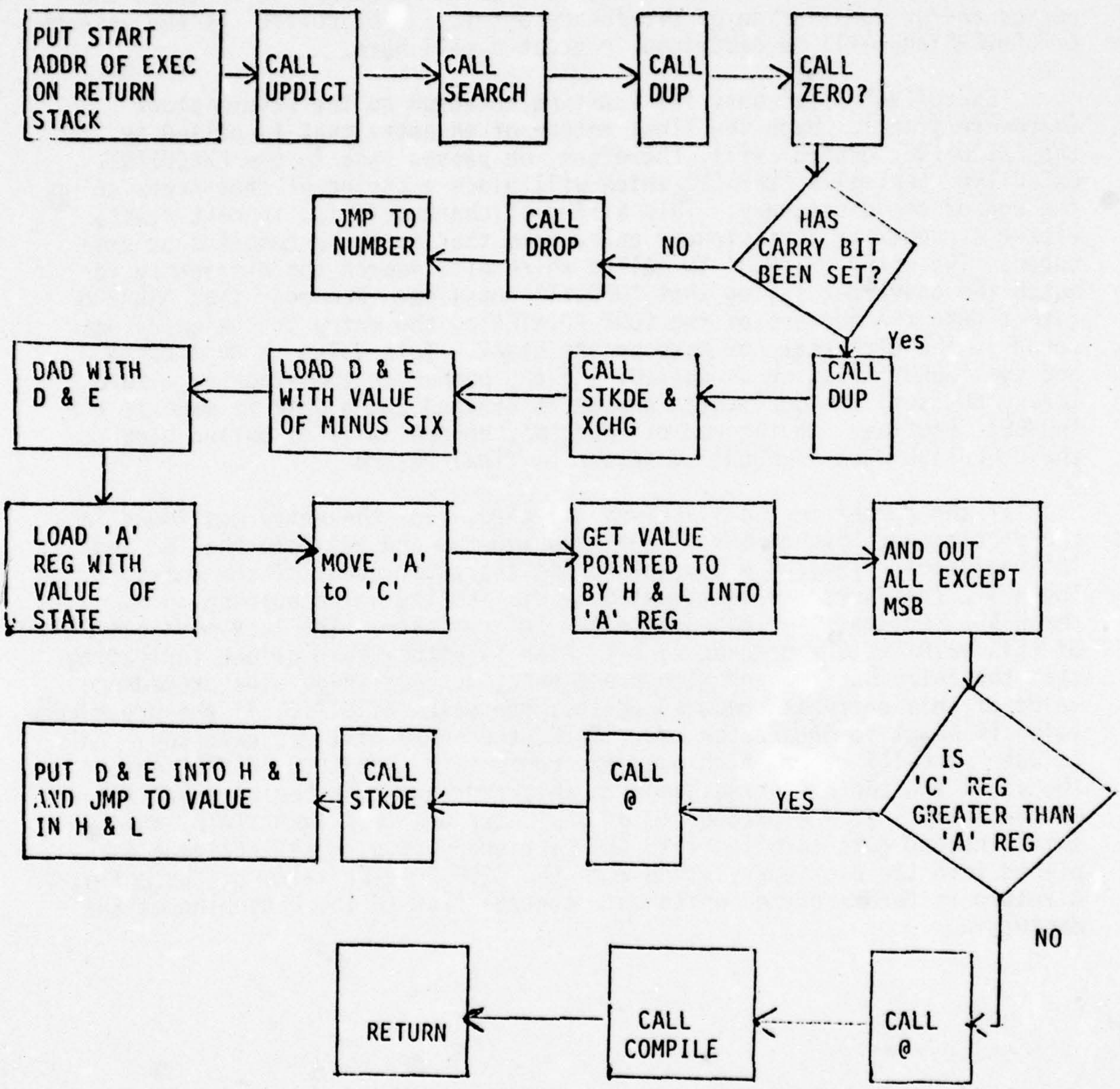
EXECUTIVE

EXECUTIVE is the main executive routine for CONVERS, it controls execution and/or compilation of dictionary entries. 'EXECUTIVE' is the essence of CONVERS and will be described in great detail here.

'EXECUTIVE' first puts its starting location on the return stack (hardware stack). Upon the final return of an entry that is called by the EXECUTIVE, control will, therefore, be passed back to the EXECUTIVE. EXECUTIVE then calls 'UPDICT' which will place a string of characters at the end of the dictionary. This string of characters is, in most cases, either a number or a dictionary entry name that is to be compiled or executed. The entry 'SEARCH' is called which will search the dictionary to match the character string that 'UPDICT' supplies. Remember that 'SEARCH' either puts the address of the CODE POINTER of the entry if the entry was found in the dictionary or zero on the stack. This value is duplicated and the 'ZERO?' routine is called. If the number on the stack is a zero, (carry not set) the duplicated number is dropped and a JUMP is made to the 'NUMBER' routine. At the end of 'NUMBER', control will be passed back to the EXECUTIVE when 'NUMBER' executes its final return.

If the number on the stack was non-zero, i.e. the entry was found in the dictionary, this number is again duplicated and put into the 'H' and 'L' registers. Thus, H & L point to the character count of the entry. The H & L registers are decremented by six and the value pointed to by the H & L registers is loaded into the 'A' register. The left-most bit of this value is the precedence bit which is either zero or one indicating that the entry has low and high precedence, respectively. The precedence value of this entry is compared against the value of STATE; if the precedence value is equal to or greater that STATE, the entry will get exectued. This is done by calling '@' which puts the contents of the CODE POINTER onto the stack and the executive jumps to this address which begins execution of the entry. If the precedence of the entry was less than STATE, the entry instead gets compiled into the dictionary, i.e. a call op-code is placed into the dictionary along with the CODE POINTER value of the entry. A return is then executed which puts control back to the beginning of the executive.

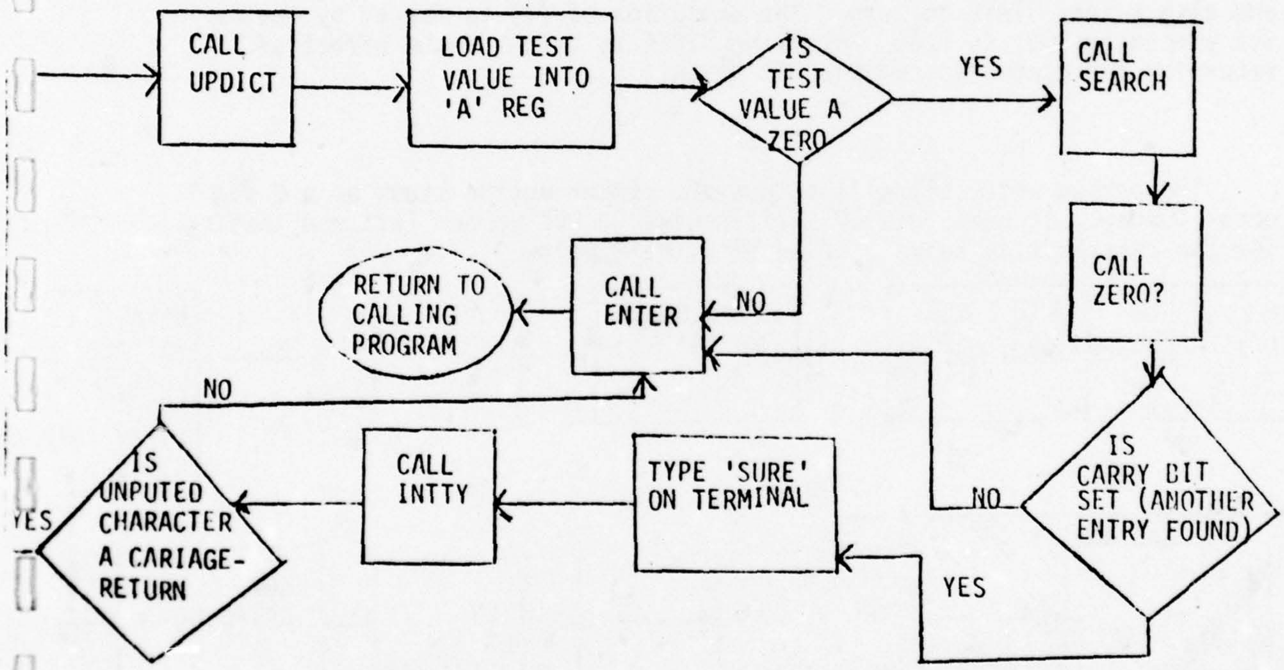
EXECUTIVE FLOW CHART


PUT START ADDR OF EXEC ON RETURN STACK
CALL UPDICT
CALL SEARCH
CALL DUP
CALL ZERO?
HAS CARRY BIT BEEN SET?
NO
CALL DROP
JMP TO NUMBER
Yes
CALL DUP
CALL STKDE & XCHG
LOAD D & E WITH VALUE OF MINUS SIX
DAD WITH D & E
LOAD 'A' REG WITH VALUE OF STATE
MOVE 'A' to 'C'
GET VALUE POINTED TO BY H & L INTO 'A' REG
AND OUT ALL EXCEPT MSB
IS 'C' REG GREATER THAN 'A' REG
YES
CALL @
CALL STKDE
PUT D & E INTO H & L AND JMP TO VALUE IN H & L
NO
CALL @
CALL COMPILE
RETURN

CODE

CODE is an entry which will begin a new entry by compiling the new entry name by calling UPDICT as well as the LINK and CODE POINTER. However, it will first check the dictionary to see if another entry with that particular name has been previously defined. (This checking process only takes place if a test variable is zero. If it is non-zero, no checking takes place; thus, to inhibit the checking process, one need only to change this value.) If another entry was found to have the same name, the message 'SURE' is typed on the terminal. 'SURE' prompts the user to type a character on the terminal; if this character is other than a carriage-return, it is assumed that the user wants to redefine the entry and calls 'ENTER'. If a carriage-return is detected, the routine goes to the beginning and calls 'UPDICT' again which allows the user to rename the entry.

CODE FLOW CHART


CALL UPDICT
LOAD TEST VALUE INTO 'A' REG
IS TEST VALUE A ZERO
YES
CALL SEARCH
CALL ZERO?
NO
RETURN TO CALLING PROGRAM
CALL ENTER
NO
IS CARRY BIT SET (ANOTHER ENTRY FOUND)
NO
YES
TYPE 'SURE' ON TERMINAL
CALL INTTY
IS UNPUTED CHARACTER A CARIAGE-RETURN
YES

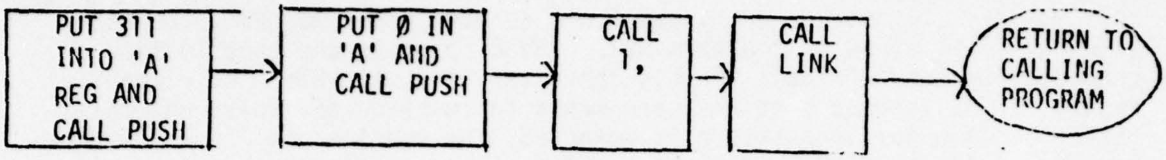
:

The entry ':' begins a colon definition which first calls 'CODE' to compile the new definition name, LINK and CODE POINTER. It then sets the variable STATE to 200_8 which has the effect of putting the system into the COMPILE mode.

RTN

RTN will end a 'CODE' definition by compiling a return op-code (311) into the entry. It also calls 'LINK' which officially 'links' the new entry into the dictionary.

RTN FLOW CHART

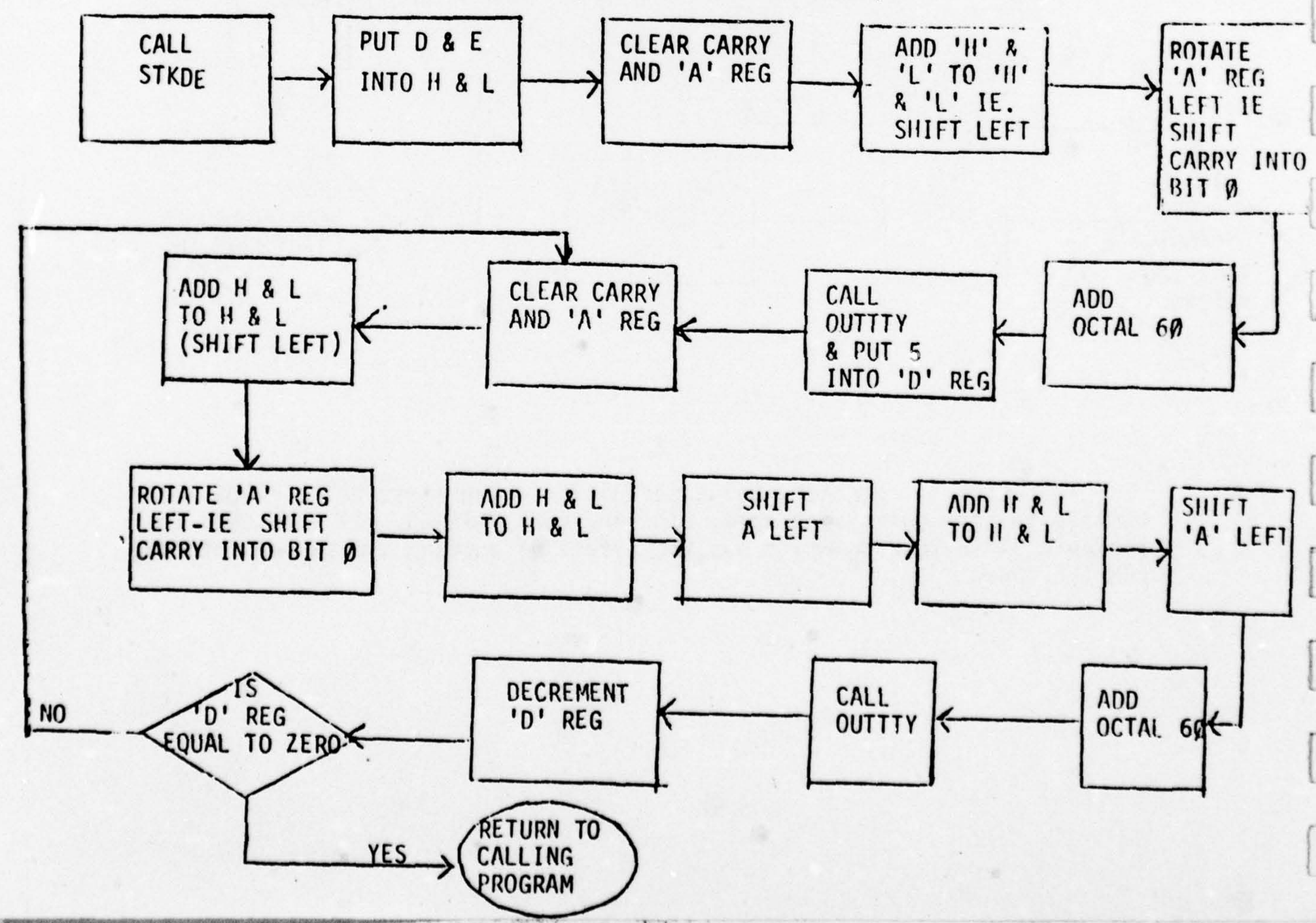

PUT 311 INTO 'A' REG AND CALL PUSH
PUT Ø IN 'A' AND CALL PUSH
CALL 1,
CALL LINK
RETURN TO CALLING PROGRAM

;

The semicolon entry (;) ends a colon (:) definition by calling 'RTN' and also resets STATE to zero. The execution of (;) is forced by having its precedence set to one. Resetting STATE to zero has the effect of returning the system to the EXECUTE state.

.

The period entry (.) will output the number on the stack as a 6 digit octal number. It does this by shifting the 16 bit number left and testing for the carry bit to be set. (See flow chart below.)


CALL STKDE
PUT D & E INTO H & L
CLEAR CARRY AND 'A' REG
ADD 'H' & 'L' TO 'H' & 'L' IE. SHIFT LEFT
ROTATE 'A' REG LEFT IE SHIFT CARRY INTO BIT Ø
ADD OCTAL 6Ø
CALL OUTTTY & PUT 5 INTO 'D' REG
CLEAR CARRY AND 'A' REG
ADD H & L TO H & L (SHIFT LEFT)
ROTATE 'A' REG LEFT-IE SHIFT CARRY INTO BIT Ø
ADD H & L TO H & L
SHIFT A LEFT
ADD H & L TO H & L
SHIFT 'A' LEFT
ADD OCTAL 6Ø
CALL OUTTTY
DECREMENT 'D' REG
IS 'D' REG EQUAL TO ZERO
NO
YES
RETURN TO CALLING PROGRAM

LITERAL,

LITERAL, is identical to LITERAL except that Part 2 of LITERAL puts the address of the data value instead of the data value itself on the stack. (See LITERAL.)

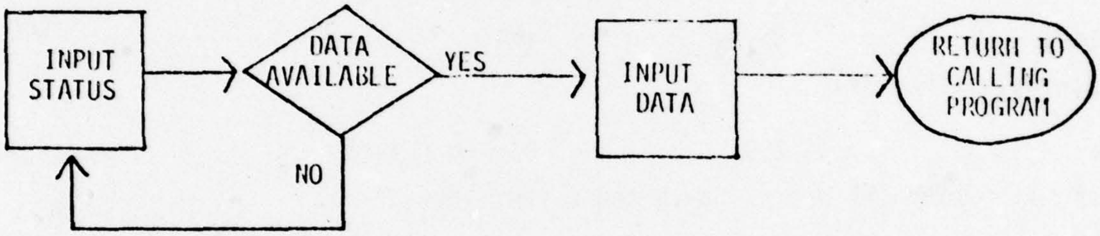
ϕϕϕϕ (NULL)

NULL simply returns to the calling program. NULL is executed when an extra space is typed between entry names; i.e., the character count is zero and all characters are zero.

INTTY

INTTY inputs a character from the terminal and places it into the 'A' register.

INTTY FLOW CHART



CONDENSED CONVERS INITIAL MACHINE CODE DICTIONARY (IMCD)

ENTRY NAME	LOCATION	EXEC TIME	REGISTER
UPDICT	114_8	Depends on Terminal	ALL

Inputs characters from terminal and places at end of dictionary until a space is encountered. The number of characters inputted is stored at the first location. If less than three characters were input, zero's are stored in the dictionary such that the # of input characters and zero's stored equal three.

INTTY	2366_8	Depends on Terminal	A

Inputs one character from terminal. (Port 1 - Status Port Ø) Ecko's character back to terminal. Strips parity from character. Leaves character in 'A' register.

OUTTTY	424_8	Depends on Terminal	A

Outputs one character to terminal. Char. to be outputted should be in 'A' register.

ENTRY NAME	LOCATION	EXEC TIME	REGISTERS
PUSH	273_8	27μsec (Fast)	H,L
Pushes contents of 'A' reg. on parameter stack. In fast mode stack overflow not checked. PUSH decrements parameter pointer by one.			
POP	321_8	27μsec (Fast)	H,L,A
Pops contents of stack into 'A' reg. In fast mode stack underflow not checked. POP increments parameter stack pointer by one.			
TTYOUT	406_8	Depends on Terminal	ALL
Outputs block of characters to terminal. 16 bit address under stack. # char (16 bits) on top of stack.			
ØØØ(NULL)	2415_8	5μsec	NONE
Does nothing. (Returns)			
STKDE	450_8	81μsec (Fast)	A,D,E,H,L
Puts 16 bit number off stack into D and E registers.			
@	471_8	175μsec (Fast)	A,D,E,H,L
Replaces address on stack with value found at address. 'Value' is a 16 bit number -- i.e., the two bytes put on stack at address and address + one.			
1@	516_8	133.5μsec (Fast)	A,D,E,H,L
Replaces address on stack with 8 bit value found at address.			
DUP	536_8	111.5μsec (Fast)	A,D,E,H,L
Duplicates 16 bit value on stack.			
DE-STK	560_8	81μsec (Fast)	A,H,L
Puts contents of D and E registers on stack.			
STK-BC	601_8	81μsec (Fast)	A,B,C,H,L
Puts 16 bit value on stack into B and C registers.			
BC-STK	622_8	81μsec (Fast)	A,H,L
Puts contents of B and C registers on stack.			

<table>
<tr><th>ENTRY NAME</th><th>LOCATION</th><th>EXEC TIME</th><th>REGISTERS</th></tr>
<tr><td>SWAP</td><td>643_8</td><td>363μsec (Fast)</td><td>ALL</td></tr>
<tr><td colspan="4">Swaps 16 bit value under stack with 16 bit value on top of stack.</td></tr>
<tr><td>HERE</td><td>670_8</td><td>107μsec (Fast)</td><td>A,D,E,H,L</td></tr>
<tr><td colspan="4">Puts contents of dictionary pointer + one on stack.</td></tr>
<tr><td>HEAD</td><td>711_8</td><td>104.5μsec (Fast)</td><td>A,D,E,H,L</td></tr>
<tr><td colspan="4">Puts contents of LINK (102_8) on stack. The LINK always contains the first address of the last definition in the dictionary.</td></tr>
<tr><td>1+</td><td>733_8</td><td>186.5μsec (Fast)</td><td>A,D,E,H,L</td></tr>
<tr><td colspan="4">Increments number on stack by one. (16 bits)</td></tr>
<tr><td>OVER</td><td>753_8</td><td>122μsec (Fast)</td><td>A,D,E,H,L</td></tr>
<tr><td colspan="4">Puts the number at the very bottom of the stack on top of the stack.</td></tr>
<tr><td>DROP</td><td>1003_8</td><td>76μsec (Fast)</td><td>A,H,L</td></tr>
<tr><td colspan="4">Removes top number from stack.</td></tr>
<tr><td>SEARCH</td><td>1022_8</td><td>Variable</td><td>ALL</td></tr>
<tr><td colspan="4">This routine searches the dictionary for the character string at the end of the dictionary (char. string deposited by UPDICT). If a match is found with one of the entries in the dict., the address of the code pointer of that entry is put on the stack. If no match is found, a zero is put on the stack.</td></tr>
<tr><td>CRLF</td><td>1147_8</td><td>Depends on Terminal</td><td>A,C</td></tr>
<tr><td colspan="4">Outputs a carriage-return and line feed.</td></tr>
<tr><td>SPACE</td><td>1172_8</td><td>Depends on Terminal</td><td>A,C</td></tr>
<tr><td colspan="4">Outputs a space.</td></tr>
<tr><td>ZERO?</td><td>1211_8</td><td>113μsec(Fast)/133μsec(Fast)</td><td>A,B,C,H,L</td></tr>
<tr><td colspan="4">This routine tests top number on stack for zero. If number is a zero, carry is reset. If number is non-zero, carry is set.</td></tr>
<tr><td>OCTCVRT</td><td>1250_8</td><td>Variable</td><td>ALL</td></tr>
<tr><td colspan="4">Converts an ASCII character string at end of dictionary (deposited by UPDICT) into a 16 bit binary number. This number is then pushed on the stack.</td></tr>
</table>

ENTRY NAME	LOCATION	EXEC TIME	REGISTERS
NUMBER	1336_8	Variable	ALL

NUMBER checks the char. string at the end of the dictionary to determine if all characters can be interpreted as octal numbers. If not, an error message is output. (? 2) If so, NUMBER calls OCTCVRT. If the state is one (COMPILE) 'LITERAL' ',' are executed. Thus, the colon definition when executed will place the converted num. on the stack.

ENTRY NAME	LOCATION	EXEC TIME	REGISTERS
!	1453_8	195.5μsec	ALL

This routine deposits the number under the stack at the address on top of the stack.

ENTRY NAME	LOCATION	EXEC TIME	REGISTERS
,	1476_8	119μsec	A,B,C,H,L

Deposits the 16 bit number on the stack into the dictionary. The Dictionary pointer is incremented by two.

ENTRY NAME	LOCATION	EXEC TIME	REGISTERS
ENTER	1534_8	89μsec	A,D,E,H,L

Begins a new entry by depositing the LINK and code pointer at the end of the dictionary. The address of the first character of the new entry is stored at a temporary LINK location. (To be used by LINK below.)

ENTRY NAME	LOCATION	EXEC TIME	REGISTERS
LINK	1613_8	21μsec	H,L

Stores the number at the temp link (which ENTER supplies) in the LINK location. (102_8) This effectively links and adds the new entry into the dictionary.

ENTRY NAME	LOCATION	EXEC TIME	REGISTERS
1,	1632_8	116.5μsec	A,B,C,H,L

Deposits the LSB 8 bits from the stack into the dictionary. Increments the Dict. pointer by one.

ENTRY NAME	LOCATION	EXEC TIME	REGISTERS
LITERAL	1656_8	300.5μsec when executed/ 443.5μsec (COMPILE)	A,D,E,H,L

When LITERAL is executed, it puts a CALL to the execution address of LITERAL. Upon execution of the CALL, the 16 bit data found after the CALL (in the entry Calling Literal) is put on the stack.

ENTRY NAME	LOCATION	EXEC TIME	REGISTERS
COMPILE	1716_8	335.5μsec	A,H,L

Puts a call opcode as well as the 16 bit number on the stack into the dictionary. This entry is used to compile a call to another routine when compiling a CODE definition.

ENTRY NAME	LOCATION	EXEC TIME	REGISTERS
EXECUTE	1752_8	up to 2μsec depending on search time	ALL

EXECUTE calls UPDICT and SEARCH -- if a zero left on the stack NUMBER is called. If not-zero (routine searched for exists in the dict.) a jump to this routine is made after putting the EXEC starting address on the return stack. If the precedence bits of the entry are less than STATE, a Call to this entry is compiled into the dictionary.

CODE	2066_8	126μsec (Fast)	ALL

Calls UPDICT and ENTER to begin a new entry at the end of the dictionary.

:	2165_8	149.5μsec (Fast)	ALL

CALLS CODE to begin a new entry, also stores a three in STATE to put system in the compile mode.

RTN	2206_8	237.5μsec (Fast)	ALL

Puts a RET opcode (311) into the dictionary and calls LINK to link the new definition into the dictionary.

;	2241_8		ALL

Calls RTN to end the definition, also changes the state variable to zero to return system to execute mode.

.	2263		A,C,D,E,H,L

Pops a number of the stack, converts and outputs a 6 digit octal number on the terminal.

LITERAL,	2332		A,D,E,H,L

Same as LITERAL except now the address of the data location is pushed on the stack instead of the data itself.

STANDARD CONVERS HIGH-LEVEL DICTIONARY

The 'Standard High-Level' CONVERS software includes most of the 'higher level' types of constructs associated with other higher level languages (such as BASIC and FORTRAN). Included are DO-LOOP's, IF-ELSE-THEN constructs, a multitude of number testing routines, and some unique routines that most people will find very unfamiliar. It should be pointed out now that the user is certainly not limited to the CONVERS system described here; a user could create new or different routines and add them to his/her system. In fact, one of the primary advantages of CONVERS is the user's ability to add (or change) any aspect of the system. However, a solid 'high-level' system should be provided for those users who do not wish to develop their own custom-tailored CONVERS software system. With this in mind, a discussion will follow on the routines that are provided in the CONVERS 'Standard High-Level' dictionary.

CONSTANT

The first routine to be defined is 'CONSTANT' which allows one to define a constant with a given name. 'CONSTANT' has been defined as follows:

```
: CONSTANT  CODE  LITERAL  ,  RTN  ;
```

The colon (:) begins execution by calling 'UPDICT' and 'ENTER'. The user would then type a character string that names the new entry, in the above case 'CONSTANT' is the entry name. Following the name of the entry, are the entry names that are to be compiled into the new entry; the semicolon (;) ends the entry. As stated initially, 'CONSTANT' allows one to define constants with any given name. For example, the user might define 'ZERO' initialized with the value zero:

```
Ø CONSTANT ZERO
```

Now, whenever ZERO is executed, a ZERO will be placed on the stack.

'CONSTANT' executes this function by first calling 'CODE' which will allow the user to input a character string that names the constant, in this case, the name is 'ZERO'. 'CONSTANT' will next call 'LITERAL' which, when executed, compiles the execution address of 'LITERAL' into 'ZERO'. 'CONSTANT' now calls the comma (,) entry which deposits the number on the stack into the next two bytes of 'ZERO'. In this case, the number 'zero' is found on the stack. 'CONSTANT' finally calls 'RTN' which places a return op-code into 'ZERO' and properly links 'ZERO' into the dictionary.

VARIABLE

'VARIABLE' allows one to define variables initialized to any value; the user can, likewise, name the variable in an analogous fashion to the way 'CONSTANT' allows the user to name a number. For example, one might define the following variable as follows:

```
1ϕϕ VARIABLE TIME
```

In the above example, 'TIME' is initialized to octal (or decimal) 1ϕϕ. However, whenever 'TIME' is executed, the address where the variable is stored is placed on the stack and not the data value itself. The reason for this will become obvious later on.

XCHG

The entry, 'XCHG', when executed, will exchange the D & E and H & L register pairs. 'XCHG' is needed since there is no direct way to load the H & L registers with any of the initial machine-code entries.

'

The apostrophe (') entry, when executed, searches the dictionary for the entry name entered after the apostrophe. For example:

```
' STKDE
```

The apostrophe will search the dictionary for the entry 'STKDE' and push on the stack the address contained at the code pointer of 'STKDE'. Thus, the apostrophe is a way of locating addresses of any dictionary entry.

JZOP

The entry 'JZOP' will, when executed, deposit a JZ op-code into the dictionary. (DP is incremented by one.) Thus, 'JZOP' is a sort of assembler mnenomic.

JMPOP

'JMPOP' puts a JMP op-code into the dictionary when executed.

JNZOP

'JNZOP' puts a JNZ op-code into the dictionary when executed.

(

The entry '(' allows one to insert comments inside of definitions as they are being compiled. All characters entered after the '(' are ignored by the system, however, they are echoed back to the terminal. The right parenthesis ')' ends the comment and returns the system back to the interpreter state. For example:

```
CODE  ZAP  (ZAP  REMOVES  NUMBER)  '  DROP  COMPILE  RTN
```

In this example, the text 'ZAP REMOVES NUMBER' is totally ignored by the system except to be echoed on the terminal.

RZ,

The entry 'RZ,' puts an RZ op-code into the dictionary.

STATE

'STATE' puts the address of the location where the variable 'STATE' is stored.

"

The quote entry (") functions identically to the parenthesis entry '(' when the system is in the execute state (STATE has the value zero). With the system in the compile state (STATE = one) the characters input after the quote will be deposited into the dictionary. Another quote (") terminates the execution of this entry.

DAD

The entry 'DAD', when executed, exchanges the D & E registers with the H & L registers and does a double byte add with the B & C registers.

+

The entry '+' adds two numbers on the stack and replaces these numbers with the sum.

1-

The entry '1-' decrements the top number on the stack by one. Later on, a subtract defintion will be defined; however, in many cases, the number need only to be decremented by one. The entry '1-' does this in a much faster fashion than a subtraction of a number by one.

COMPLEMENT

The 'COMPLEMENT' entry complements the top number on the stack.

MINUS

The entry 'MINUS' does a two's complement of the number on the stack, i.e. it complements and increments the number by one.

ØPUSH

The entry 'ØPUSH' pushes an 8-bit zero on the stack. Since all stack operators require a 16 bit number on the stack, 'ØPUSH' is used extensively when inputting values from I/O devices which interface through an 8-bit port.

-

The entry '-' subtracts the top number on the stack from the bottom number and replaces the two numbers with the difference.

0<

The entry '0<' tests the number on the stack to determine if it is less than zero, i.e. is the MSB of the number a one. If the number is negative, a 'one' is put on the stack; otherwise, a zero is put on the stack. The entry '0<' replaces the original number on the stack as do most stack operators.

PRECEDENCE

The entry 'PRECEDENCE' will change the precedence value of any entry to three. This has the effect of forcing execution of the entry even when the system is in the compile state. As an example:

```
PRECEDENCE '
```

In the above example, 'PRECEDENCE' is used to change the precedence value of the apostrophe (') entry.

STK-DICT

'STK-DICT' is an imperative definition (precedence is one) that will compile a call to 'LITERAL' as well as deposit the number on the stack into any type of defintion (COLON or CODE). Other than through 'STK-DICT', there is no way of compiling a number on the stack into a colon definition during the compilation of the new entry.

IF, ELSE and THEN

The entries 'IF', 'ELSE' and 'THEN' taken together are the backbone of the conditional execution construct of CONVERS. These entries work as follows:

```
: TEST  IF  BELL  ELSE  CRLF  THEN  ;
```

In the 'TEST' example above, if the number on the stack is other than zero, the routines between the 'IF' and 'ELSE' entries will be executed. (In the above example, the terminal bell would ring.) If the number on the stack is a zero, the entries between the 'ELSE' and 'THEN' would be executed. Finally, any entries after the 'THEN' are executed. The 'ELSE' entry is not needed to properly compile the IF-THEN construct.

=

The entry '=' compares the two numbers on top of the stack for equality; if they are equal, a one is put on the stack, otherwise, a zero. The two original numbers are removed.

>

The entry '>' tests for a greater than or equal condition, i.e. is the bottom number on the stack greater than or equal to the top number. If true, a one is put back on the stack, otherwise, a zero.

<

The entry '<' tests for a less than or equal condition, i.e. is the bottom number less than or equal to the top number. If true, a one is put back on the stack, otherwise a zero.

BELL

'BELL' will ring the terminal bell.

DO & LOOP

The 'DO' and 'LOOP' entries (conditional looping) make up the most useful construct of any software system. The respective CONVERS structure, however, works somewhat differently than the analogous FORTRAN or BASIC DO-LOOP structure. An example of the way this structure works in CONVERS will follow:

```
: SOUND  5  1  DO  BELL  LOOP  ;
```

In the above example, 'SOUND' has been defined to ring the terminal bell 5 times. Note that the routines that are to be iteratively executed are entered between the 'DO' and 'LOOP' entries as the definition is compiled. 'SOUND' in the above example supplies the upper and lower indices (the numbers 5 & 1 respectively) inside the definition with the lower index always being the top number on the stack. To make 'SOUND' more general, it should be defined as follows:

```
: SOUND  DO  BELL  LOOP  ;
```

In this example, the upper and lower indices must be put on the stack prior to the execution of 'SOUND'. Some other definition might supply these indices or the user may as well:

```
1Ø  1  SOUND
```

After the user typed the above, ten (octal or decimal) bells will sound on the terminal.

DADSP

This entry adds the stack pointer to the H & L registers when executed. By zeroing the H & L registers beforehand, DADSP will allow one to copy the current SP value into the H & L registers. Given the 8080 instruction set, this is the only way to access the stack pointer.

IC

'IC' puts the current value of the stack pointer (hardware) on the software (parameter) stack.

I, J & K

'I', 'J' and 'K', respectively, put the current value of the inner, next, and outer DO-LOOP lower indices on the stack. This allows for nesting of DO-LOOP's three levels deep and still being able to access the lower index of each loop. By copying the compilation of these routines, entries may be written to allow access of the lower index at any level.

AND

'AND' will logically AND two numbers on the stack replacing the two numbers with the result.

DP

'DP' will push on the stack the address of the dictionary pointer.

,CODE

The entry ',CODE' works identically to 'CODE' except that the code pointer will be incremented by the number found on top of the stack. This allows local constants to be inserted at the start of the defintion without interfering with the execution of the rest of the defintion. (Remember that a defintion will be executed or compiled at the address contained in the code pointer of the defintion.)

REMEMBER

The entry 'REMEMBER' allows the user to define an entry (using REMEMBER) such that at a later time the user can 'erase' everything in the dictionary after (and including the entry) by simply typing this entry name on the terminal. This also allows 'application dictionaries' to be swapped in and out from mass storage by 'erasing' this dictionary after terminating its function.

2*

The entry '2*' multiplies the number on the stack by two. (It does a one bit left shift.)

2/

The entry '2/' divides the number on the stack by two by performing a one bit right shift.

SWITCH

The entry 'SWITCH' switches the high and low 8 bits of the number on the stack.

SP

The entry 'SP' puts the current value of the software stack pointer on the stack. The SP value is 'read' before it is put on the stack.

UNDER

'UNDER' adds a number on the stack to the current SP and fetches the number under the stack pointed to by the above sum. This number is then pushed on top of the stack. For example, 3 UNDER will copy and push the third number under the stack on top of the stack.

MAX

'MAX' replaces two numbers on the stack with the maximum of the two numbers.

MIN

'MIN' replaces two numbers on the stack with the minimum of two numbers.

1!

The entry '1!' deposits the 8 LSB's of the number under the stack at the address on top of the stack.

1WORD

The entry '1WORD' will copy one byte from the address underneath the stack to the location on top of the stack. '1WORD' increments these two addresses and leaves them on the stack.

WORDS

'WORDS' copies a block of memory from one location to another. The number of bytes to be transferred should be on top of the stack, the destination address underneath that and the source address underneath the destination address.

CVRT

'CVRT' is a specialized routine that is used for decimal conversion.

.10

The entry '.10' converts and outputs the top number on the stack in decimal.

1ØCVRT

The entry '1ØCVRT' will convert the character string at the end of the dictionary (that 'UPDICT' supplies) into a binary number and pushes it onto the stack. '1ØCVRT' assumes that the character string represents a decimal number.

DECIMAL

The 'DECIMAL' entry sets the number conversion system to the decimal mode, i.e. all entries will be input and output in decimal.

OCTAL

'OCTAL' returns the system to the octal mode, i.e. all number conversions will be in octal.

DATAOUT

'DATAOUT' will output the contents of the 'A' reg. to a specified port. The entry 'PORTOUT?' will supply the port address.

PORTOUT?

'PORTOUT?' will take the number on top of the stack and compile it into the 'DATAOUT' routine such that 'DATAOUT' will output the 'A' register contents to the correct port.

OUTDEVICE

'OUTDEVICE' will output the 8 LSB's of the number under the stack to the port addressed by the 8 LSB's of the number on top of the stack.

INDATA

'INDATA' will input to the 'A' register the contents of a specified input port. The 'PORTIN?' routine will supply the port address.

PORTIN?

'PORTIN?' will take the number on the stack and compile it into the 'INDATA' routine such that 'INDATA' will input data from the correct port.

INDEVICE

'INDEVICE' will input and push onto the stack the 8 bits from the port address found on the stack.

SPECIAL

'SPECIAL' is a special variable location that is used in the 'BEGIN-HERE' routine below.

BEGIN-HERE

'BEGIN-HERE' is an imperative definition (it is always executed independent of 'STATE') that stores DP + 1 into the variable location 'SPECIAL'. 'BEGIN-HERE' is used like 'HERE' ('HERE' is used in a similiar fashion for compiling 'CODE' definitions); it stores the current address of memory during the compilation of a colon (:) definition. This address would be typically used later in the definition to 'JMP' back to the earlier address upon some condition.

BEGIN

'BEGIN' compiles a 'JMP' to the address contained at the variable 'SPECIAL' into the dictionary. It is used in conjunction with 'BEGIN-HERE' to allow backward referencing in a colon (:) defintion. 'BEGIN' is an imperative defintion.

END

'END' is another imperative definition that will compile a return op-code (311_8) into the dictionary. This allows for unconditional termination of an entry. 'END' is usually used in the context of the 'IF-ELSE-THEN' construct to conditionally terminate an entry.

THE INITIAL MACHINE CODE DICTIONARY (IMCD)

(The IMCD object tape supplied is assembled at a starting address of 1ØØ$_8$.)

```
LABEL       OPCODE          OPERAND             COMMENTS

DP            DW            KEND+ 243_10        ; DICT. POINTER
LINK          DW            ELINK               ; LINK
LINK1         DB            6                   ; # CHAR
              DB            'U'                 ; NAME OF UPDICT
              DB            'P'                 ; ROUTINE
              DB            'D'
              DW            Ø                   ; LINK
              DW            $ + 2               ; CODE POINTER
```

```
LABEL       OPCODE      OPERAND         COMMENTS

UPDCT       XRA         A               ; THIS ROUTINE
            LHLD        DP              ; WILL INPUT CHAR
            INX         H               ; STRING & PUT IN
            MOV         M,A             ; DICT UNTIL SPACE
            INX         H               ; DETECTED. FIRST
            MOV         M,A             ; LOCAT CONTAINS #
            INX         H               ; CHAR INPUTTED
            MOV         M,A
            INX         H
            MOV         M,A
            DCX         H
            DCX         H
            NOP
            MOV         D,A
HEREA       CALL        INTTY
            CPI         16_8
            JC          HEREA
            CPI         4Ø_8
            JZ          STOP
            INR         D
            MOV         M,A
            INX         H
            JMP         HEREA
STOP        LHLD        DP
            INX         H
            MOV         M,D
            RET
TEST        PUSH        PSW
            PUSH        D
            LHLD        DP
            LXI         D, 1Ø_8
            DAD         D
            MOV         A,L
            CMA
            MOV         E,A
```

```
LABEL        OPCODE       OPERAND              COMMENTS

             MOV          A,H
             CMA
             MOV          D,A
             INX          D
             LHLD         SP
             DAD          D
             JNC          ERR1
             POP          D
             POP          PSW
             RET
ERR1         MVI          B, 63$_8$
ERR          MVI          A, 77$_8$
             CALL         OUTTY
             MVI          A, 4Ø$_8$
             CALL         OUTTY
             MOV          A,B
             CALL         OUTTY
             XRA          A
             STA          STATE
             JMP          EXEC
LINK2        DB           5
             DB           'D'
             DB           'U'
             DB           'M'
             DW           LINK1
             DW           $ + 2
DUMMY        CALL         NULL
             RET
             NOP
LINK3        DB           4
             DB           'P'
             DB           'U'
             DB           'S'
             DW           LINK2
             DW           $ + 2
```

```
LABEL       OPCODE      OPERAND         COMMENTS

PUSH        CALL        TEST
            LHLD        SP
            DCX         H
            MOV         M,A
            SHLD        SP
            RET
SP          DW          SPA
LINK4       DB          3
            DB          'P'
            DB          'O'
            DB          'P'
            DW          LINK3
            DW          $ + 2
POP         LHLD        SP
            DAD         H
            MOV         A,H
            STC
            CMC
            RAR
            INR         A
            CPI         POP?
            JC          OKAY
            MVI         B, 61$_8$
            JMP         ERR
OKAY        LHLD        SP
            MOV         A,M
            INX         H
            SHLD        SP
            RET
LINK5       DB          6
            DB          'T'
            DB          'T'
            DB          'Y'
            DW          LINK4
            DW          $ + 2
TTYOT       CALL        STKDE
            CALL        STKBC
            INR         D
```

```
LABEL       OPCODE      OPERAND            COMMENTS
AGAIN       LDAX        B
            CALL        OUTTY
            INX         B
            DCR         E
            JNZ         AGAIN
            DCR         D
            JNZ         AGAIN
            RET
            NOP
            NOP
LINK6       DB          6
            DB          'O'
            DB          'U'
            DB          'T'
            DW          LINK5
            DW          $ + 2
OUTTY       PUSH        PSW
HEREZ       IN          STATU
            ANI         TBE
            JZ          HEREZ
            POP         PSW
            OUT         TTY
            RET
LINK7       DB          5
            DB          'S'
            DB          'T'
            DB          'K'
            DW          LINK6
            DW          $ + 2
STKDE       CALL        POP
            MOV         D,A
```

```
LABEL             OPCODE           OPERAND                    COMMENTS
                    CALL             POP
                    MOV              E,A
                    RET
LINK8               DB               1
                    DB               '@'
                    DB               Ø
                    DB               Ø
                    DW               LINK7
                    DW               $ + 2
D@                  CALL             STKDE
                    LDAX             D
                    CALL             PUSH
                    INX              D
                    LDAX             D
                    CALL             PUSH
                    RET
LINK9               DB               2
                    DB               'i'
                    DB               '@'
                    DB               Ø
                    DW               LINK8
                    DW               $ + 2
DI@                 CALL             STKDE
                    LDAX             D
                    CALL             PUSH
                    RET
LNK10               DB               3
                    DB               'D'
                    DB               'U'
                    DB               'P'
```

```
LABEL          OPCODE         OPERAND              COMMENTS

                 DW           LINK9
                 DW           $ + 2
DUP              CALL         STKDE
                 CALL         DESTK
                 CALL         DESTK
                 RET
LNK11            DB           6
                 DB           'D'
                 DB           'E'
                 DB           '_'
                 DW           LNK1Ø
                 DW           $ + 2
DESTK            MOV          A,E
                 CALL         PUSH
                 MOV          A,D
                 CALL         PUSH
                 RET
LNK12            DB           6
                 DB           'S'
                 DB           'T'
                 DB           'K'

                 DW           LNK11
                 DW           $ + 2
STKBC            CALL         POP
                 MOV          B,A
                 CALL         POP
                 MOV          C,A
                 RET
LNK13            DB           6
```

```
LABEL          OPCODE       OPERAND              COMMENTS
               DB           'B'
               DB           'C'
               DB           '_'
               DW           LNK12
               DW           $ + 2
BCSTK          MOV          A,C
               CALL         PUSH
               MOV          A,B
               CALL         PUSH
               RET
LNK14          DB           4
               DB           'S'
               DB           'W'
               DB           'A'
               DW           LNK13
               DW           $ + 2
SWAP           CALL         STKBC
               CALL         STKDE
               CALL         BCSTK
               CALL         DESTK
               RET
LNK15          DB           4
               DB           'H'
               DB           'E'
               DB           'R'
               DW           LNK14
               DW           $ + 2
HERE           LHLD         DP
               INX          H
               XCHG
               CALL         DESTK
               RET
```

```
LABEL       OPCODE      OPERAND          COMMENTS
LNK16       DB          4
            DB          'H'
            DB          'E'
            DB          'A'
            DW          LNK15
            DW          $ + 2
HEAD        LHLD        LINK
            XCHG
            CALL        DESTK
            RET
            NOP
            NOP
LNK17       DB          2
            DB          '1'
            DB          '+'
            DB          Ø
            DW          LNK16
            DW          $ + 2
D1+         CALL        STKDE
            INX         D
            CALL        DESTK
            RET
LNK18       DB          4
            DB          'O'
            DB          'V'
            DB          'E'
            DW          LNK17
            DW          $ + 2
OVER        LHLD        SP
            INX         H
            INX         H
            INX         H
```

```
LABEL          OPCODE        OPERAND                  COMMENTS

               INX           H
               MOV           D,M
               INX           H
               MOV           E,M
               CALL          DESTK
               RET
               NOP
               NOP
LNK19          DB            4
               DB            'D'
               DB            'R'
               DB            'O'
               DW            LNK18
               DW            $ + 2
DROP           CALL          POP
               CALL          POP
               RET
LNK20          DB            6
               DB            'S'
               DB            'E'
               DB            'A'
               DW            LNK19
               DW            $ + 2
SERCH          LHLD          LINK
               SHLD          TEMP
               XCHG
               LHLD          DP
               INX           H
HERE5          MVI           C,4_8
```

```
LABEL       OPCODE      OPERAND         COMMENTS

HERE1       LDAX        D
            ANI         177
            CMP         M
            JNZ         NEXT
            INX         H
            INX         D
            DCR         C
            JNZ         HERE1
            INX         D
            INX         D
            NOP
HERE2       CALL        DESTK
            RET
NEXT        LHLD        TEMP
            XCHG
            INX         D
            INX         D
            INX         D
            INX         D
            LDAX        D
            SUI         Ø
            MOV         L,A
            INX         D
            LDAX        D
            JZ          AGN
HERE6       NOP
            MOV         H,A
            SHLD        TEMP
            XCHG
            LHLD        DP
            INX         H
            JMP         HERE5
```

```
LABEL         OPCODE      OPERAND              COMMENTS

AGN           SUI          Ø
              JNZ          HERE6
              MOV          H,A
              XCHG
              JMP          HERE2
              NOP
              NOP
TEMP          NOP
              NOP
LNK21         DB           4
              DB           'C'
              DB           'R'
              DB           'L'
              DW           LNK2Ø
              DW           $ + 2
CRLF          MVI          A,15$_8$
              CALL         OUTTY
              MVI          A,12$_8$
              CALL         OUTTY
              RET
LNK22         DB           5
              DB           'S'
              DB           'P'
              DB           'A'
              DW           LNK21
              DW           $ + 2
SPACE         MVI          A,4Ø$_8$
              CALL         OUTTY
              RET
LNK23         DB           5
              DB           'Z'
```

```
LABEL       OPCODE      OPERAND         COMMENTS

            DB          'E'
            DB          'R'
            DW          LNK22
            DW          $ + 2
ZERO?       CALL        STKBC
            MVI         A,Ø
            CMP         B
            JNZ         STC
            CMP         C
            JNZ         STC
            XRA         A
            RET
            NOP
            NOP
            NOP
STC         STC
            RET
            NOP
            NOP
            NOP
LNK24       DB          7
            DB          'O'
            DB          'C'
            DB          'T'
            DW          LNK23
            DW          $ + 2
CVT         LHLD        DP
            INX         H
            MOV         B,M
```

```
LABEL         OPCODE     OPERAND              COMMENTS

              XCHG
              XRA         A
              MOV         H,A
              MOV         L,A
              MVI         A,5
              CMP         B
              JNC         CVT1
              INX         D
              LDAX        D
              SUI         6Ø8
              RLC
              RLC
              RLC
              MOV         L,A
              MVI         B,5
CVT1          INX         D
              LDAX        D
              SUI         6Ø8
              ADD         L
              MOV         L,A
              DCR         B
              JZ          CVT2
              DAD         H
              DAD         H
              DAD         H
              JMP         CVT1
CVT2          XCHG
              CALL        DESTK
              RET
```

```
LABEL       OPCODE      OPERAND          COMMENTS
LNK25       DB          6
            DB          'N'
            DB          'U'
            DB          'M'
            DW          LNK24
            DW          $ + 2
NUMBR       CALL        HERE
            CALL        D@
            CALL        HERE
            CALL        D1+
            CALL        STKDE
            CALL        STKBC
NUM1        LDAX        D
            CPI         $60_8$
            JC          NMERR
            CPI         $70_8$
            JNC         NMERR
            INX         D
            DCR         C
            JNZ         NUM1
            CALL        CVT
            LDA         STATE
            CPI         0
            RZ
            CALL        LITRL
            CALL        D,
            RET
NMERR       LXI         D, ERBF
            CALL        DESTK
            LXI         D,3
            CALL        DESTK
```

```
LABEL     OPCODE        OPERAND              COMMENTS
          CALL          TTYOT
          RET
ERBF      DB            '?'
          DB            40_8
          DB            '2'
LNK26     DB            1
          DB            '!'
          DW            0
          DW            LNK25
          DW            $ + 2
D!        CALL          STKDE
          CALL          STKBC
          XCHG
          MOV           M,C
          INX           H
          MOV           M,B
          RET
LNK27     DB            1
          DB            ','
          DW            0
          DW            LNK26
          DW            $ + 2
D,        CALL          STKBC
          LHLD          DP
          INX           H
          MOV           M,C
          INX           H
          MOV           M,B
          SHLD          DP
          RET
          DW            0
```

```
LABEL       OPCODE      OPERAND             COMMENTS

            DW          0
            DW          0
            DW          0
LNK28       DB          5
            DB          'E'
            DB          'N'
            DB          'T'
            DW          LNK27
            DW          $ + 2
ENTER       LHLD        DP
            INX         H
            SHLD        ETEMP
            INX         H
            INX         H
            INX         H
            INX         H
            XCHG
            LHLD        LINK
            MOV         A,L
            STAX        D
            INX         D
            MOV         A,H
            STAX        D
            INX         D
            MOV         H,D
            MOV         L,E
            INX         H
            INX         H
            MOV         A,L
            STAX        D
            INX         D
            MOV         A,H
```

```
LABEL       OPCODE      OPERAND             COMMENTS
            STAX        D
            XCHG
            SHLD        DP
            RET
ETEMP       DW          0
            DW          0
LNK29       DB          4
            DB          'L'
            DB          'I'
            DB          'N'
            DW          LNK28
            DW          $ + 2
DLINK       LHLD        ETEMP
            SHLD        LINK
            RET
LNK30       DB          2
            DB          '1'
            DB          ','
            DB          0
            DW          LNK29
            DW          $ + 2
D1,         CALL        STKBC
            LHLD        DP
            INX         H
            MOV         M,C
            SHLD        DP
            RET
LNK31       DB          7
            DB          'L'
            DB          'I'
```

```
LABEL     OPCODE        OPERAND          COMMENTS
          DB            'T'
          DW            LNK30
          DW            $ + 2
LITRL     LXI           D,LITHR
          CALL          DESK
          CALL          COMPL
          RET
LITHR     NOP
          POP           H
          MOV           D,H
          MOV           E,L
          INX           H
          INX           H
          PUSH          H
          CALL          DESTK
          CALL          D@
          RET
LNK32     DB            7
          DB            'C'
          DB            'O'
          DB            'M'
          DW            LNK31
          DW            $ + 2
COMPL     MVI           A,315$_8$
          CALL          PUSH
          MVI           A,0
          CALL          PUSH
          CALL          D1,
          CALL          D,
          RET
STATE     DB            0
          DW            0
```

```
LABEL     OPCODE     OPERAND             COMMENTS

LNK33     DB         11_8,
          DB         'E'
          DB         'X'
          DB         'E'
          DW         LNK32
          DW         $ + 2
EXEC      LXI        SP,KEND+240_10
          LXI        H,EXEC
          PUSH       H
          CALL       UPDCT
          CALL       SERCH
          CALL       DUP
          CALL       ZERO?
          JC         EXEC1
          CALL       DROP
          JMP        NUMBR
EXEC1     CALL       DUP
          CALL       STKDE
          XCHG
          LXI        D,177772
          DAD        D
          LDA        STATE
          MOV        C,A
          MOV        A,M
          ANI        200
          NOP
          NOP
          CMP        C
          JC         EXEC2
```

```
LABEL      OPCODE      OPERAND          COMMENTS
           CALL        D@
           CALL        STKDE
           XCHG
           PCHL
EXEC2      CALL        D@
           CALL        COMPL
           RET
           NOP
           NOP
           NOP
LNK34      DB          4
           DB          'C'
           DB          'O'
           DB          'D'
           DW          LNK33
           DW          $ + 2
CODE       CALL        UPDCT
           LDA         TSTVL
           CPI         0
           JNZ         FORGT
           CALL        SERCH
           CALL        ZERO?
           JC          SURE
FORGT      CALL        ENTER
           RET
SURE       LXI         D,SUREA
           CALL        DESTK
           LXI         D,4
           CALL        DESTK
           CALL         TTYOT
           CALL         INTTY
           CPI         15$_8$
```

```
LABEL        OPCODE        OPERAND                COMMENTS

             JNZ           FORGT
             JMP           CODE
SUREA        DB            'S'
             DB            'U'
             DB            'R'
             DB            'E'
TSTVL        DB            0
LNK35        DB            1
             DB            ':'
             DW            0
             DW            LNK34
             DW            $ + 2
             CALL          CODE
             MVI           A,2ØØ
             STA           STATE
             RET
LNK36        DB            3
             DB            'R'
             DB            'T'
             DB            'N'
             DW            LNK35
             DW            $ + 2
RTN          MVI           A,311_8
             CALL          PUSH
             MVI           A,0
             CALL          PUSH
             CALL          D1,
             CALL          DLINK
             RET
             DW            0
```

```
LABEL      OPCODE     OPERAND          COMMENTS
LNK37       DB        1
            DB        ';'
            DW        0
            DW        LNK36
            DW        $ + 2
D;          CALL      RTN
            MVI       A,0
            STA       STATE
            RET
            NOP
LNK38       DB        1
            DB        '.'
            DW        0
            DW        LNK37
            DW        $ + 2
D.          CALL      STKDE
            XCHG
            XRA       A
            DAD       H
            RAL
            ADI       608
            CALL      OUTTY
            MVI       D,5
LOOP        XRA       A
            DAD       H
            RAL
            DAD       H
            RAL
            DAD       H
            RAL
```

```
LABEL      OP.CODE     OPERAND         COMMENTS
           ADI         60_8
           CALL        OUTTY
           DCR         D
           JNZ         LOOP
           RET
LNK39      DB          10_8
           DB          'L'
           DB          'I'
           DB          'T'
           DW          LNK38
           DW          $ + 2
LTRL,      LXI         D,LR,
           CALL        DESTK
           CALL        COMPL
           RET
LR,        POP         H
           MOV         D,H
           MOV         E,L
           INX         H
           INX         H
           PUSH        H
           CALL        DESTK
           RET
LNK40      DW          0
           DW          0
           DW          LNK39
           DW          $ + 2
NULL       RET
ELINK      DB          5
           DB          'I'
```

LABEL	OPCODE	OPERAND	COMMENTS
	DB	'N'	
	DB	'T'	
	DW	LNK40	
	DW	$ + 2	
FLAG?	IN	STATU	
	ANI	DAV	
	JZ	FLAG?	
	IN	TTY	
	ANI	177_8	
	CALL	OUTTY	
	RET		
KEND	NOP		
STATU	EQU	0	;STATUS PORT
TTY	EQU	1	;I/O PORT
TBE	EQU	200_8	;TBE FLAG
DAV	EQU	100_8	;DAV FLAG
SPA	ORG	20000_8	;UPPER MEM BOUND+1
POP?	EQU	041_8	;MSB OF BOUND+2

472:GAN:716:tam
78u472-608

TECHNICAL REPORT DISTRIBUTION LIST, GEN

No. Copies

Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217
Attn: Code 472 — 2

ONR Branch Office
536 S. Clark Street
Chicago, Illinois 60605
Attn: Dr. George Sandoz — 1

ONR Branch Office
715 Broadway
New York, New York 10003
Attn: Scientific Dept. — 1

ONR Branch Office
1030 East Green Street
Pasadena, California 91106
Attn: Dr. R. J. Marcus — 1

ONR Area Office
One Hallidie Plaza, Suite 601
San Francisco, California 94102
Attn: Dr. P. A. Miller — 1

ONR Branch Office
Building 114, Section D
666 Summer Street
Boston, Massachusetts 02210
Attn: Dr. L. H. Peebles — 1

Director, Naval Research Laboratory
Washington, D.C. 20390
Attn: Code 6100 — 1

The Assistant Secretary
of the Navy (R,E&S)
Department of the Navy
Room 4E736, Pentagon
Washington, D.C. 20350 — 1

Commander, Naval Air Systems Command
Department of the Navy
Washington, D.C. 20360
Attn: Code 310C (H. Rosenwasser) — 1

No. Copies

Defense Documentation Center
Building 5, Cameron Station
Alexandria, Virginia 22314 — 12

U.S. Army Research Office
P.O. Box 1211
Research Triangle Park, N.C. 27709
Attn: CRD-AA-IP — 1

Naval Ocean Systems Center
San Diego, California 92152
Attn: Mr. Joe McCartney — 1

Naval Weapons Center
China Lake, California 93555
Attn: Dr. A. B. Amster
Chemistry Division — 1

Naval Civil Engineering Laboratory
Port Hueneme, California 93401
Attn: Dr. R. W. Drisko — 1

Professor K. E. Woehler
Department of Physics & Chemistry
Naval Postgraduate School
Monterey, California 93940 — 1

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D.C. 20380 — 1

Office of Naval Research
800 N. Quincy Street
Arlington, Virginia 22217
Attn: Dr. Richard S. Miller — 1

Naval Ship Research and Development
Center
Annapolis, Maryland 21401
Attn: Dr. G. Bosmajian
Applied Chemistry Division — 1

Naval Ocean Systems Center
San Diego, California 91232
Attn: Dr. S. Yamamoto, Marine
Sciences Division — 1

Encl 1

TECHNICAL REPORT DISTRIBUTION LIST, 051C

No.
Copies

Dr. M. B. Denton
University of Arizona
Department of Chemistry
Tucson, Arizona 85721 1

Dr. R. A. Osteryoung
Colorado State University
Department of Chemistry
Fort Collins, Colorado 80521 1

Dr. B. R. Kowalski
University of Washington
Department of Chemistry
Seattle, Washington 98105 1

Dr. S. P. Perone
Purdue University
Department of Chemistry
Lafayette, Indiana 47907 1

Dr. D. L. Venezky
Naval Research Laboratory
Code 6130
Washington, D.C. 20375 1

Dr. H. Freiser
University of Arizona
Department of Chemistry
Tuscon, Arizona 85721

Dr. Fred Saalfeld
Naval Research Laboratory
Code 6110
Washington, D.C. 20375 1

Dr. E. Chernoff
Massachusetts Institute of Technology
Department of Mathematics
Cambridge, Massachusetts 02139 1

No.
Copies

Dr. K. Wilson
University of California, San Diego
Department of Chemistry
La Jolla, California 1

Dr. A. Zirino
Naval Undersea Center
San Diego, California 92132 1

Dr. John Duffin
United States Naval Postgraduate School
Monterey, California 93940 1

Dr. G. M. Hieftje
Department of Chemistry
Indiana University
Bloomington, Indiana 47401 1

Dr. Victor L. Rehn
Naval Weapons Center
Code 3813
China Lake, California 93555 1

Dr. Christie G. Enke
Michigan State University
Department of Chemistry
East Lansing, Michigan 48824 1

Dr. Kent Eisentraut, MBT
Air Force Materials Laboratory
Wright-Patterson AFB, Ohio 45433 1

Walter G. Cox, Code 3632
Naval Underwater Systems Center
Building 148
Newport, Rhode Island 02840 1